(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
31 July 2003 (31.07.2003)

PCT

(10) International Publication Number
WO 03/063427 A1

(51) International Patent Classification⁷: H04L 12/56, H04Q 11/04

(21) International Application Number: PCT/SE03/00064

(22) International Filing Date: 16 January 2003 (16.01.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
0200153-5 21 January 2002 (21.01.2002) SE

(71) Applicant *(for all designated States except US)*: **ROCKSTORM TECHNOLOGIES AB** [SE/SE]; Aurorum 7, S-977 75 Luleå (SE).

(72) Inventor; and
(75) Inventor/Applicant *(for US only)*: **SUNDSTRÖM, Mikael** [SE/SE]; Regnvägen 24, S-976 32 Luleå (SE).

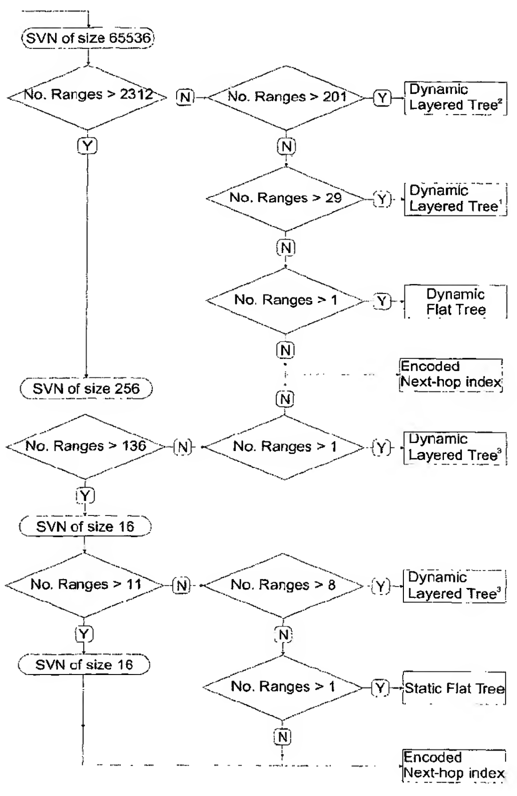
(74) Agent: **HOPFGARTEN, Nils**; Groth & Co., KB, P.O. Box 6107, S-102 32 Stockholm (SE).

(81) Designated States *(national)*: AE, AG, AL, AM, AT (utility model), AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ (utility model), CZ, DE (utility model), DE, DK (utility model), DK, DM, DZ, EC, EE (utility model), EE, ES, FI (utility model), FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK (utility model), SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, SE, SI,

[Continued on next page]

(54) Title: METHOD AND SYSTEM FOR FAST IP ROUTING LOOKUP USING FORWARDING TABLES WITH GUARANTEED COMPRESSION RATE AND LOOKUP PERFORMANCE



(57) Abstract: A method for IP routhing lookup to determine where to forward an IP -datagram with a given destination address by retrieving from a routing table a next/hop index indicating where to forward said datagram, said next/hop index being associates with the longest matching prefix of said destination address, said address being a number in an address universe U, whereing a set of address prefixes P and a mapping of P onto a set of next/hop indices D are converted into a set of ranges R, constituting a partition of U, and a mapping of R onto D. The method involves the steps of building and storing in a memory a forwarding table representation from R and D by using a predetermined layered data structure where the construction of the layer is selected depending on the range density |R´| for the sub-universe U´represented by that layer to get a space efficient representation of the set of ranges R, and performing the lookup by a range matching operation in said forwarding table. A corresponding system comprises a first converting means for converting a set of address prefixes P into a set of ranges R constituting a partition of said universe U and a second converting means for converting the mapping from P onto a set of next-hop indices D to an equivalent mapping from R onto D. The system also comprises data structuring means for forming predetermined layered datastructures T representing the routing table, and building and memory means for building and storing a forwarding tablerepresentation from R and D by using a predetermined layered data structure where the construction of the layer is selected depending on the range density |R´| for the sub-universe U´ represented by that layer to get a space efficient representation of the set of ranges R, and means for performing the lookup by a range matching operation in said forwarding table.

SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declaration under Rule 4.17:
— *of inventorship (Rule 4.17(iv)) for US only*

Published:
— *with international search report*

Method and system for fast IP routing lookup using forwarding tables with guaranteed compression rate and lookup performance

1 Technical Field

The present invention relates to a method and a system for fast IP routing lookup, in a compressed forwarding table, to determine where to forward an IP-datagram with a given destination address by retrieving from a routing table a next-hop index indicating where to forward said datagram, said next-hop index being associated with the longest matching prefix of said destination address, said address being a number in an address universe $\mathbf{U}$, wherein a set of address prefixes $\mathbf{P}$ and a mapping of $\mathbf{P}$ onto a set of next-hop indices D are converted into a set of ranges $\mathbf{R}$, constituting a partition of $\mathbf{U}$, and a mapping of $\mathbf{R}$ onto D. The system according to the invention comprises a first converting means for converting said set of address prefixes $\mathbf{P}$ into a set of ranges $\mathbf{R}$ constituting a partition of said universe $\mathbf{U}$ and a second converting means for converting the mapping from $\mathbf{P}$ onto D to an equivalent mapping from $\mathbf{R}$ onto D.

2 Background

Internet is formed of a plurality of networks connected to each other, wherein each of the constituent networks maintains its identity. Each network supports communication among devices connected to the networks, and the networks in their turn are connected by routers. Thus, Internet can be considered to comprise a mass of routers interconnected by links. Communication among nodes (routers) on Internet takes place using an Internet protocol, commonly known as IP. IP datagrams (packets) are transmitted over links from one router to the next one on their ways towards the final destinations. In each router a forwarding decision is performed on incoming datagrams to determine the datagrams next-hop router.

A routing or forwarding decision is normally performed by a lookup procedure in a routing table. Thus, IP routers do a routing lookup in the routing table to obtain next-hop information about where to forward the IP datagrams on their path toward their destinations. A routing lookup operation on an incoming datagram requires the router to find the most specific path for the datagram. This means that the router has to solve the so-called longest prefix matching problem which is the problem of finding the next-hop information (or index) associated with the longest address prefix matching the incoming datagrams destination address in the set of arbitrary length (i.e. between 0 and 32 bits) prefixes constituting the routing table.

To speed up the forwarding decisions, many IP router designs of today use a caching technique wherein the most recently or most frequently looked up destination addresses and the corresponding routing lookup results are kept in a route cache. This method works quite well for routers near the edges of the network, i.e. so called small office and home office (SOHO) routers, that have small routing tables, low traffic loads, and high locality of accesses in the routing table. Another method of speeding up the routers is to exploit the fact that the frequency of routing table updates, resulting from topology changes in the network etc., is extremely low compared to the frequency of routing lookups. This makes it feasible to store the relevant information from the routing table in a more efficient forwarding table optimized for supporting fast lookups. When changes to the routing table occurs, the forwarding table is partially or completely rebuilt.

In *P Gupta Algorithms for Routing Lookups and Packet Classification*, A Dissertation Submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University, December 2000, a forwarding table representation and a corresponding lookup procedure using merely Static Vector Nodes, i.e. direct addressing, in two steps (i.e. 2 memory accesses) is described. With a continuing increase of the use of Internet there is a constant need of improving the IP routing lookup. In WO 01/22667 a technique is described for making direct indexing more efficient by compressing vector nodes having few used buckets. In the described system a forwarding table consisting of three levels of compressed pointer arrays are used and the cost for a lookup is 6 memory accesses in the worst case. In case of sparsely populated intervals the waste of memory space is considerable when using both these methods, even though the second is much better. The so called Luleå Algorithm is another forwarding table, described in WO 99/14906, where the memory space utilization is heavily improved but the cost for lookup is increased to 12 memory accesses.

The purpose of the present invention is to render the IP routing still more efficient by introducing a new forwarding table representation that improves the memory space utilization even further while reducing the cost for a lookup to 4 memory accesses.

3 Disclosure of the Invention

This purpose is obtained by a method and a system of the kind defined in the introductory portion of the description and having the characterizing features of Claims 1 and 18 respectively. In the present invention a series of data structures is consequently developed for representing the forwarding table which data structures are adapted to the interval density of the subuniverse in question. Thus an optimum structure is selected for a certain interval density in order to minimize the memory needs. For each memory access a certain block size is introduced and starting from this size the data structure is adapted such that the memory access will be as efficient as possible. There is no problem to reduce the number of memory accesses for lookup if there is no limit for the available memory and, on the other hand, there is no problem to perform a compression of the table if the number of required memory accesses for lookup can be disregarded. With the present invention both these quantities are minimized. According to another aspect of the present invention a computer program product is provided having computer program code means to make a computer execute the above method when the program is run on a computer.

4 Brief Description of the Drawings

To further explain the invention embodiments chosen as examples will now be described in greater details with reference to the drawings on which Figure 1 is a flowchart showing the hierarchy of the different sub-data-structures and the corresponding lookup procedures according to the invention, Figure 2 illustrates next-hop index and sub-tree pointer encoding scheme used in a first step static vector nodes, Figure 3 illustrates a Dynamic Flat Tree sub-data structure with 25 ranges showing the layout of the information with respect to cache line boundaries, Figure 4 illustrates a Dynamic Layered Tree leaf and the encoding of range boundaries to next hop indices as well as the corresponding node, Figure 5 further illustrates the layout of information in a binary Dynamic Layered Tree data structure, Figure 6 shows the layout of information in a Static Flat Tree data structure, Figure 7 is a schematic block diagram of an embodiment of the system according to the invention, and Figure 8 shows an example of a small forwarding table represented by a 32 bits variation of the Dynamic Layered Tree data structure.

5 Description of Preferred Embodiments

In the following the development of a series of data structures used to represent compressed forwarding tables supporting fast routing lookups is described. In particular, we develop numerous techniques that are combined to compress 2^{18} routes into a forwarding table data structure using less than 2.7 Mbytes of memory and still not requiring more than 4 memory accesses for lookup.

5.1 Preliminaries

5.1.1 Some Notes on Notation

In the following we will express ranges or intervals by using the dot notation — for numbers a and b, $a < b$, we will denote ranges or sequences (depending on the context) involving a and b as follows

$$\begin{aligned} b-1\ldots b &= b-1, b \\ a\ldots b &= a, a+1\ldots b. \end{aligned}$$

We will also use various entities represented by bit strings. Unless explicitly stated otherwise, the bits are numbered starting from 0 and ending with the highest bit. Let x be a $k+1$ bits bit string. Bit i is denoted by $x[i]$ and the whole bit string can be written as

$$\begin{aligned} x &= x[k]\,x[k-1]\ldots x[1]\,x[0] \\ &= x[k\ldots 0]. \end{aligned}$$

The reason for starting with the highest bit is that we often consider the bit strings as non-negative integers where the bit string constitutes the binary representation. The size of a bit string x is denoted by $|x|$, i.e. for x in the previous example we have that $|x| = k+1$. We can append bit strings by simply placing them after each other — $z = xy$ is the bit string obtained when appending x to y. That is, $z[|y|-1\ldots 0] = y$ and $z[|y|+|x|-1\ldots |y|] = x$. The string obtained by repeatedly appending k copies of x is denoted by x^k, i.e.

$$x^k = \underbrace{xx\ldots x}_{k \text{ times}}$$

Finally, we will represent sets such as $X = \{x_1, x_2, \ldots, x_n\}$. The size of X is denoted by $|X|$ and in this case, $|X| = n$. In addition, we will use the standard set relations such as $\in$, $\subset$, $\cap$, $\cup$, and $\subseteq$ in the following ways

$$\begin{aligned} x_1 &\in X \\ \{x_1, x_2, \ldots, x_{n-1}\} &\subset X \\ X &\subseteq X \\ \{x_0, x_1, \ldots, x_n\} \cap \{x_1, x_2, \ldots, x_{n+1}\} &= X \\ \{x_1, x_2, \ldots, x_i\} \cup \{x_{i+1}, x_{i+2}, \ldots, x_n\} &= X. \end{aligned}$$

5.1.2 Routing Tables and Routing Lookups

A *routing table* is a *surjective mapping* from a set of *IP-address prefixes*, or simply *address prefixes*, to a set of *next-hop* indices. Routing tables are used to determine where to forward an *IP-datagram* with a given *destination address*, as mentioned above. This is achieved by finding the *longest matching prefix* of the destination address and retrieve the associated next-hop index representing what to do with the packet. The process of performing the longest prefix match and retrieving the next-hop index is referred to as *routing lookup*.

An *IPv4-address* is a number in the universe

$$\mathbf{U} = \left\{0, 1, \ldots, 2^{32} - 1\right\}.$$

It is represented by a bit vector

$$a = a[31]\,a[30]\ldots a[1]\,a[0] = a[31\ldots 0]$$

equivalent to the binary representation of the number. Address prefixes are also represented by bit vectors. The prefix of length 0 is denoted by the wildcard symbol $*$ and matches any address. Any other prefix, of non-zero length k, is denoted by

$$p = \underbrace{p[31]\,p[30]\ldots p[i]}_{k \text{ bits}} * = p[31\ldots 32-k] *$$

and matches any address a satisfying

$$a[31\ldots 32-k] = p[31\ldots 32-k].$$

Address prefixes also represents *subsets* of $\mathbf{U}$ consisting of ranges. That is, the prefix $p\,[31\ldots 32-k]*$ represent the range $p^{lo}\ldots p^{hi}$ where the binary representation of p^{lo} and p^{hi} are given by

$$p\,[31\ldots 32-k]\,\underbrace{00\ldots 0}_{32-k \text{ bits}}$$

and

$$p\,[31\ldots 32-k]\,\underbrace{11\ldots 1}_{32-k \text{ bits}}$$

respectively. For any pair of prefixes P_1 and P_2 we have either $P_1 \subset P_2$, $P_1 = P_2$, or $P_1 \supset P_2$. It is rather straight forward to convert the set of prefixes to a set of n ranges

$$\mathbf{R} = \{R_1,\, R_2,\, \ldots,\, R_n\}$$

constituting a *partition* of $\mathbf{U}$, where the first, ith, and last range are given by

$$\begin{aligned} R_1 &= r_1^{lo}\ldots r_1^{hi} = 0\ldots r_1^{hi}, \\ R_i &= r_i^{lo}\ldots r_i^{hi} = r_{i-1}^{hi}+1\ldots r_i^{hi}, \end{aligned}$$

and

$$R_n = r_n^{lo}\ldots r_n^{hi} = r_{n-1}^{hi}+1\ldots 2^{32}-1$$

respectively. Moreover, each range will be associated with a next-hop index in the same way as the original prefixes, i.e. we have a surjective mapping from $\mathbf{R}$ to the set of next-hop indices. Thus, we can perform routing lookup by finding the *only matching range* of an address a, i.e. the R_i satisfying $r_i^{lo} \le a \le r_i^{hi}$, instead of performing the longest prefix match. The conversion can be performed as follows. Initially, let $\mathbf{R}$ be empty and let the *current address* a be 0. Then repeatedly insert $R = r^{lo}\ldots r^{hi}$, where $r^{lo} = a$ and r^{hi} is the smallest address larger than or equal to r associated with the same next-hop index as a, into $\mathbf{R}$ and assign $r^{hi}+1$ to a as long as $a \in \mathbf{U}$.

We conclude this section with a summary of the steps performed so far and a more formal definition of the problems yet to solve. Let $D = \{d_1, d_2, \ldots, d_m\}$ be the set of next-hop indices (the letter D/d is chosen to represent the looked up *data*). Moreover, let next-hop index d_i be represented by a $\lceil \lg m \rceil$ bits non-negative integer

$$\underbrace{d_i\,[\lceil \lg m\rceil - 1]\, d_i\,[\lceil \lg m\rceil - 2]\ldots d_i\,[1]\, d_i\,[0]}_{\lceil \lg m\rceil \text{ bits}}.$$

Initially, we had a set of prefixes **P** and a mapping of **P** onto D. By the longest prefix match operation we also have a mapping of **U** onto D. Finally, we have described a method for converting **P** and the mapping of **P** onto D to a set of ranges **R**, constituting a partition of **U**, and a mapping of **R** onto D. In effect, we have converted the original longest prefix match problem in **P** to the simpler but equivalent (only) range matching problem in **R**.

The conversion described can be applied to any routing table representation to retrieve this partition of ranges (and corresponding next-hop indices) serving as an intermediate representation from which the forwarding table is built (see Figure 7 bp2 and bp3). The core of the present invention:

- the efficient forwarding table representation constructed from the intermediate range partition representation and
- the lookup procedure used to retrieve the next-hop index associated with only matching range from said intermediate representation

can therefore be used in conjunction with *any* routing table implementation to accelerate the routing lookups.

5.1.3 Computational Model

The worst case, from the point of view of costs, for performing a computation such as a routing lookup in a fairly large data structure is mainly dependent on the number of memory accesses performed. In each memory access, a cache line or block of k consecutive bytes is accessed from the main memory. The block is then stored at all levels of the cache hierarchy before the actual data is accessed in the fastest *first level cache*. Since the dominating cost for a memory access is the copying of the block from main memory, we count multiple accesses to the same k bytes block during a short computation such as a lookup as one memory access.

5.1.4 Design Parameters

In our target architecture, e.g. the Intel Pentium III, each cache line consists of 32 bytes and this is the value of k we will use throughout the design of the algorithm.

It is estimated that the size of the largest routing tables will exceed 200000 routes in the next few years. High-end routing table data structures and implementations of today are therefore designed to accommodate for $2^{18} = 262144$ routes and the same number of next-hop indices. Hence, we need to use 18 bits to represent a next-hop index. The conversion described in the previous section converts n prefixes to at most $2n + 1$ ranges. Therefore, our data structure is designed to store $2 \cdot 2^{18} + 1 = 524289$ ranges. Our design goal is to perform a routing lookup in at most 4 memory accesses.

5.2 Compressed Forwarding Table Representation

The basic principle behind the compressed forwarding table is to repeatedly reduce the size of the sub-universes in up to four steps corresponding to the design parameter of 4 memory accesses. In the present invention, the size of each sub-problem instance (i.e. number of ranges) with respect to the size of the sub-universe is carefully observed to determine the best approach for representing the ranges in that particular sub-universe.

5.2.1 Overview of the Data Structure

In order to get a better understanding of the basic data structure used in the present invention and the following principles for size optimization, we will assume for now that the method of repeated reduction is used exclusively.

Initially, we have a set of n ranges $\mathbf{R} = \{R_1, R_2, \ldots, R_n\}$ constituting a partition of the original universe $\mathbf{U}$. The first step is to partition $\mathbf{U}$ into 2^{16} sub-universes $\mathbf{U}_0$, $\mathbf{U}_1, \ldots, \mathbf{U}_{2^{16}-1}$ of size 2^{16} each. This is achieved by starting with 2^{16} empty sets of 16 bits ranges $\mathbf{R}_0, \mathbf{R}_1, \ldots, \mathbf{R}_{2^{16}-1}$ and repeating the following until $\mathbf{R}$ is empty. Let $R = r^{lo} \ldots r^{hi} \in \mathbf{R}$ and $s = r^{lo}[31 \ldots 16]$. If

$$r^{lo}[31 \ldots 16] = r^{hi}[31 \ldots 16]$$

then remove R from $\mathbf{R}$ and add $r^{lo}[15 \ldots 0] \ldots r^{hi}[15 \ldots 0]$ to $\mathbf{R}_s$. Otherwise, add

$$r^{lo}[15 \ldots 0] \ldots \left(2^{16} - 1\right)$$

to $\mathbf{R}_s$ and replace R by $2^{16} \cdot (s + 1) \ldots r^{hi}$ in $\mathbf{R}$. Upon completion of the procedure, $\mathbf{R}$ is empty and the n 32 bits ranges are converted into n or more 16 bits ranges distributed among the 2^{16} range sets. For each $\mathbf{R}_i$ containing only one range, let T_i be the next-hop index of that range. Otherwise, let T_i be the data structure (not yet described) representing the ranges of $\mathbf{R}_i$ and supporting the 16 bit range matching operation for looking up the next-hop index. By organizing pointers to T_0, $T_1, \ldots,$ and $T_{2^{16}-1}$ in a *Static Vector Node*, or pointer array, of size 2^{16}, we obtain a data structure supporting complete 32 bit range matching. Given an IPv4 address a, we first extract the 16 most significant bits $a[31\ldots16]$ and use these to index into the array to retrieve $T_{a[31\ldots16]}$. We then lookup $a[15\ldots0]$ in $T_{a[31\ldots16]}$ using the 16 bit range matching operation supported by the sub-data structure.

Let $|\mathbf{R}_i| > 1$ and consider the representation of T_i. By applying the same idea as above, we partition $\mathbf{U}_i$ into 2^8 sub-universes $\mathbf{U}_{i,0}$, $\mathbf{U}_{i,1}$, $\ldots$, $\mathbf{U}_{i,2^8-1}$. This is followed by processing the set of ranges $\mathbf{R}_i$ into $\mathbf{R}_{i,0}$, $\mathbf{R}_{i,1}$, $\ldots$, $\mathbf{R}_{i,2^8-1}$ and representing these by sub-data structures

$$T_{i,0},\ T_{i,1},\ \ldots,\ T_{i,2^8-1}.$$

Range matching in T_i is achieved by using $a[15\ldots8]$ to lookup $T_{i,a[15\ldots8]}$ by direct indexing in the static vector node, followed by looking up $a[7\ldots0]$ in $T_{i,a[15\ldots8]}$.

We apply the same idea on the $T_{i,j}$s by partitioning each $\mathbf{U}_{i,j}$ into 2^4 sub-universes

$$\mathbf{U}_{i,j,0},\ \mathbf{U}_{i,j,1},\ \ldots,\ \mathbf{U}_{i,j,2^4-1}.$$

The resulting range sets, the $\mathbf{R}_{i,j,k}$s, are represented by data structures

$$T_{i,j,0},\ T_{i,j,1},\ \ldots,\ T_{i,j,2^4-1}.$$

Lookup is achieved by using $a[7\ldots4]$ to index into the static vector node containing pointers to the sub-data structures. The result is either a next-hop index or an array of 2^4 next-hop indices which is indexed, using the 4 least significant bits $a[3\ldots0]$ or the IPv4 address, to complete the lookup.

Level	$\lvert\mathbf{U}\rvert$	min $\lvert\mathbf{R}\rvert$	Size	Address bits	Pointer size (bits)
1	2^{32}	∞	2^{16}	$31\ldots16$	32
2	2^{16}	2313	2^8	$15\ldots8$	20
3	2^8	137	2^4	$7\ldots4$	20
4	2^4	12	2^4	$3\ldots0$	18

In the table above, we give a summary of the figures related to the repeated partitioning and usage of static vector nodes. For a given level, $|\mathbf{U}|$ is the size of the sub-universe represented at that level. When introducing space optimizations, we will use alternative and more space efficient approaches instead of static vector nodes if the number of ranges is less than min $|\mathbf{R}|$. The size column shows the number of pointers and the Address bits column shows which bits of the IPv4 address are being used for indexing at each level. Depending on the level and alignment conditions for the various sub-structures we will use, the Pointer size, or number of bits required for representing a pointer, varies somewhat between the levels. For example, we only need 18 bits at the lowest level since we can guarantee that the result from the lookup is a next-hop index.

In what follows, we give an overview of the various sub-data structures we will use and in the next few sections we will describe each kinds of structure in detail.

Lvl	$\|\mathbf{R}\|_{\min}^{\max}$	Structure	Height	Byte alignment
2	$2 \ldots 29$	Dynamic Flat Tree	$1 \ldots 3$	4
2	$30 \ldots 201$	Dynamic Layered Tree[1]	$1 \ldots 3$	8
2	$201 \ldots 2312$	Dynamic Layered Tree[2]	$1 \ldots 3$	32
2	$2313 \ldots 2^{16}$	Static Vector Node	N/A	2048/3
3	$2 \ldots 136$	Dynamic Layered Tree[3]	$1 \ldots 2$	32
3	$137 \ldots 2^{8}$	Static Vector Node	N/A	128/3
4	$2 \ldots 8$	Dynamic Layered Tree[3]	1	32
4	$9 \ldots 11$	Static Flat Tree	1	32
4	$12 \ldots 2^{4}$	Static Vector Node	N/A	256/7

The first column shows the level of the data structure in question. Level 1 is the top level, i.e. the static vector node with 2^{16} pointers. At each level, we will determine which data structure to use based on the number of ranges (see Figure 1). For each level and data structure we show the range density interval, in the $|\mathbf{R}|_{\min}^{\max}$ column, for which that data structure offers the most space efficient representation. In the third column, we show the name of the data structure and the fourth column contains the height of the data structure measured in the number of memory accesses required to perform the lookup. Observe that we use three slightly different kinds of dynamic layered trees. All data structures starting at level 2 are at most 3 levels high except for the static vector node where the concept of height is not directly applicable. This means that no matter which of these we use, no more than 3 memory accesses is spent for completing the lookup after the first memory access at level 1. In the same way, the data structures starting at level 3 are at most two levels high and the ones starting at level 4 are at most 1 level high. Hence, we will never use more that 4 memory accesses for a complete lookup. In the last column we show the byte alignment of the various structures (for static vector nodes, we show alignment for 3 and 7 node packages respectively). The alignment is important in the following discussion regarding pointer sizes and pointer space utilization for static vector nodes.

Figure 1 is a flowchart illustrating the hierarchy of the different data sub-structures and the corresponding lookup procedures according to this embodiment of the invention. Level 1 in the figure is a Static Vector Node (SVN) of size $2^{16} = 65536$ pointers. If the number of ranges $|\mathbf{R}|$ at level 2 does not exceed 2312 one of the sub-data structures Dynamic Layered Tree[1], Dynamic Layered Tree[2] or Dynamic Flat Tree is used for representing the set of ranges and the corresponding lookup procedures are used for completing the lookup. Thus, if $201 < |\mathbf{R}| < 2312$ the Dynamic Layered Tree[1] sub-data structure representation is used. If $29 < |\mathbf{R}| < 201$ the sub-data structure Dynamic Layered Tree[2] is used, and if $1 < |\mathbf{R}| < 29$ the binary sub-data structure Dynamic Flat Tree is used. Otherwise, if $|\mathbf{R}| > 2312$ at level 2, a static vector node SVN lookup is performed and the lookup continues at level 3. If the number of ranges $|\mathbf{R}|$ at level 3 does not exceed 136 the Dynamic Layered Tree[3] sub-data structure representation is used. Otherwise for $136 < |\mathbf{R}| < 2^8$ (= 256) a SVN lookup is performed and the lookup continues at level 4. Finally, at level 4, if the number of ranges $|\mathbf{R}|$ does not exceed 11 one of the sub-data structures Dynamic Layered Tree[3] or Static Flat Tree is used for completing the lookup procedure. Thus, if $8 < |\mathbf{R}| < 11$ the binary sub-data structure Dynamic Layered Tree[3] is used, and if $1 < |\mathbf{R}| < 8$ the sub-data structure Static Flat Tree is used. Otherwise, for $11 < |\mathbf{R}| < 16$ an SVN is used for completing the lookup and obtaining the desired enclosed next-hop index.

5.2.2 Static Vector Nodes

The level 1 or top level data structure consists of a root node with 2^{16} potential sub-trees. The 16 most significant bits of the IPv4 address are used to index into the array representing the root node to extract a 32 bits non-negative integer that either encodes an index into the next-hop table or a pointer to a sub-data structure as shown in the table above. As mentioned above, we will use different kinds of sub-trees depending on the number of range boundaries that lies in the sub-universe, i.e. the density of the sub-universe. The encoding of next-hop indices and sub-tree pointers is described in Figure 2, which shows the next-hop index and sub-tree kind and pointer encoding scheme used in level 1 Static Vector Nodes.

Observe that for densities in the range $2 \ldots 201$ we must encode the size of the sub-tree exactly to accommodate for quantization reduction space optimizations. Such optimizations are not required when the density exceeds 201 (encoded as 202).

At level 2 and 3 we will use 20 bits pointers in the Static Vector Nodes. We reserve the two most significant bits for encoding the pointer kind. The remaining bits are sufficient to encode 18 bits next-hop indices as well as 18 bits pointers referring to Dynamic Layered Trees, Static Flat Trees, or Static Vector Nodes. These data structures are 32 bytes aligned. This means that 18 bits is sufficient to address a memory area of $2^{18} \cdot 32 = 8388608$ bytes.

The 20 bits pointers used at level 2 are not so easily aligned. However, since the only requirement on a vector node is that it is possible to index among the pointers and retrieve *one pointer in one memory access*, we are able to use some tricks to achieve a space efficient representation. In one cache line, we can fit $\left\lfloor \frac{256}{20} \right\rfloor = 12$ whole 20 bits pointers. By using $21 + 1/3$ cache lines, we can store $(21 + 1/3) \cdot 12 = 256$ pointers to represent a static vector node at level 2 (see Figure 1). To improve alignment conditions, we pack level 2 nodes together in groups of three. Each package requires $(21 + 1/3) \cdot 3 \cdot 32 = 2048$ consecutive bytes of storage resulting in a byte alignment of $2048/3$ (2048 bytes block per package of 3 nodes).

We can use the same approach do represent the level 3 nodes. By using $1 + 1/3$ cache lines, we can store $(1 + 1/3) \cdot 12 = 16$ pointers to represent a Static Vector Node at level 3 as appears from Figure 1. The alignment is improved in the same fashion as above, by again packing the nodes together in groups of 3. At this level, each package requires $(1 + 1/3) \cdot 3 \cdot 32 = 128$ bytes resulting in a byte alignment of $128/3$.

At the lowest level, i.e. level 4, the pointers in the static vector nodes *must* represent next-hop indices. Therefore, we only need 18 bits pointers at this level. In one cache line, we can fit $\left\lfloor \frac{256}{18} \right\rfloor = 14$ whole 18 bits pointers. By using $1 + 1/7$ cache lines, we can represent the $(1 + 1/7) \cdot 14 = 16$ pointers required for a complete level 4 node. We pack the nodes together in groups of 7 to achieve better alignment. Each package occupies a memory area of $(1 + 1/7) \cdot 7 \cdot 32 = 256$ bytes, resulting in a byte alignment of 256/7.

5.2.3 Dynamic Flat Trees

Let $\mathbf{R} = \{R_1, R_2, \ldots, R_n\}$ be the set of ranges and $d_1, d_2, \ldots, d_n$ be the associated next-hop indices constituting the current sub-problem ($2 \leq n < 29$). Moreover, let $r_1, r_2, \ldots, r_{n-1}$ be the sorted list of ranges boundaries obtained by taking the r^{lo} from each range in $\mathbf{R}$. By this construction, we have that d_1 is associated with $0..r_1$, d_2 is associated with $r_1 + 1..r_2$, and so on until finally d_n is associated with $r_{n-1} + 1..\infty$. Recall that each d_i is an 18 bits non-negative integer and let h_i and l_i be the 2 most significant bits of d_i and the 16 least significant bits of d_i respectively. Moreover, let

$$H = h_1 h_2 \ldots h_n 0^k,$$

where k is the minimum number of zeroes that needs to be appended in order achieve

$$|H| \equiv 0 \pmod{16}.$$

After packing the high 2 bits of the next-hop indices and padding H with zeroes, the information we need to represent consists of

$$n - 1 + n + \left\lceil \frac{n}{8} \right\rceil = 2n - 1 + \left\lceil \frac{n}{8} \right\rceil$$

16 bits blocks. To improve the alignment conditions, we want to use an even number of 16 bits blocks. Therefore, we add one empty block if $2n - 1 + \left\lceil \frac{n}{8} \right\rceil$ is odd. We then get a total of

$$u_{16}(n) = 2 \cdot \left\lceil \frac{2n - 1 + \left\lceil \frac{n}{8} \right\rceil}{2} \right\rceil$$

16 bits blocks.

Let $B_1, B_2, \ldots, B_{u_{16}(n)}$ denote the information blocks and $S_1, S_2, \ldots, S_{u_{16}(n)}$ denote the memory slots that will contain the information blocks. For $n = 16$ we have

$$\begin{aligned} u_{16}(16) &= 2 \cdot \left\lceil \frac{2 \cdot 16 - 1 + \left\lceil \frac{16}{8} \right\rceil}{2} \right\rceil \\ &= 34 \\ &= 16 + 16 + 2. \end{aligned}$$

Since 32 blocks corresponds to exactly 2 cache lines and 16 bits blocks are packaged in pairs these 34 blocks can not be distributed over more than 3 cache lines. It follows that the search cost is bounded by 3 memory accesses no matter how we store the information blocks in the memory slots. Therefore, we store B_1 in S_1, B_2 in S_2, ..., and $B_{u_{16}(n)}$ in $S_{u_{16}(n)}$ whenever $n \leq 16$.

For $n = 17$ we have

$$\begin{aligned} u_{16}(17) &= 2 \cdot \left\lceil \frac{2 \cdot 17 - 1 + \left\lceil \frac{17}{8} \right\rceil}{2} \right\rceil \\ &= 36 \\ &= 2 + 16 + 16 + 2. \end{aligned}$$

If we do not pay attention to the storage of information blocks in the memory slots and the location of the memory slots with respect to cache line boundaries, the information may be distributed over 4 cache lines resulting in a search cost of 4 memory accesses.

Let S_i be the first memory slot located in the beginning of a cache line. For $n \geq 17$ we store B_1 in S_i, B_2 in S_{i+1}, B_3 in S_{i+2}, ..., $B_{u_{16}(n)-(i-1)}$ in $S_{u_{16}(n)}$, $B_{u_{16}(n)-(i-2)}$ in S_1, $B_{u_{16}(n)-(i-3)}$ in S_2, ..., and $B_{u_{16}(n)}$ in S_{i-1}.

The actual information is finally stored in the information blocks as follows:

- H is stored in $B_1 \ldots B_{\lceil \frac{n}{8} \rceil}$
- $r_1 \ldots r_{n-1}$ is stored in $B_{\lceil \frac{n}{8} \rceil + 1} \cdots B_{\lceil \frac{n}{8} \rceil + n - 1}$
- $l_1 \ldots l_n$ is stored in $B_{\lceil \frac{n}{8} \rceil + n} \cdots B_{\lceil \frac{n}{8} \rceil + 2n - 1}$

By accessing the first cache line containing $B_1 \ldots B_{16}$ we can extract the 2 most significant bits of the resulting next-hop index and also search among at the first range boundaries. In the second memory access, we complete the search among the range boundaries in $B_{17} \ldots B_{32}$. The third memory access is used to extract the 16 least significant bits of the resulting next-hop index.

In Figure 3 we show an example of the layout of the information with respect to cache line boundaries. The Figure illustrates a Dynamic Flat Tree sub-data structure containing 25 ranges. Observe that the last 16 least significant bits of the last three next-hop indices l_{23}, l_{24}, and l_{25} are stored in the end of the first cache line. The lookup will start by accessing the second cache line in use.

5.2.4 Dynamic Layered Trees

Dynamic Layered Trees consists of two building blocks, *leafs* and *nodes*. Leafs consists of up to 8 next-hop indices and up to 7 range boundaries packed in a cache line and nodes consists of up to 16 range boundaries packed in a cache line. Figure 4 illustrates the Dynamic Layered Tree leaf and the encoding of range boundaries and next-hop indices in the upper part of the figure, and the corresponding node containing only range boundaries in the lower part of the figure.

Let $r_1, r_2, \ldots, r_{n-1}$ be a sorted list of range boundaries and $d_1, d_2, \ldots, d_n$ be the corresponding list of next-hop indices. As above, d_1 is associated with $0..r_1$, d_2 is associated with $r_1 + 1..r_2$, and so on until finally d_n is associated with $r_{n-1} + 1..\infty$.

For $n = 8$, we construct a tree simply by storing the range boundaries and next-hop indices in a leaf as shown in Figure 4.

For $n = 136 = 17 \cdot 8$, the tree consists of 17 leafs, each representing 8 ranges, and 1 node representing 16 range boundaries. The first leaf contains $r_1, r_2, \ldots, r_7$ and $d_1, d_2, \ldots, d_8$, the second leaf contains $r_9, r_{10}, \ldots, r_{15}$ and $d_9, d_{10}, \ldots, d_{16}$, and so on until the last leaf which contains $r_{129}, r_{130}, \ldots, r_{135}$ and $d_{129}, d_{130}, \ldots, d_{136}$. That is, the ith leaf contains

$$r_{8 \cdot (i-1)+1}, r_{8 \cdot (i-1)+2}, \ldots, r_{8 \cdot (i-1)+7}$$

and

$$d_{8 \cdot (i-1)+1}, d_{8 \cdot (i-1)+2}, \ldots, d_{8 \cdot (i-1)+8}.$$

The node contains $r_8, r_{16}, \ldots, r_{8i}, \ldots, r_{128}$. By searching the node in 1 memory access, we can determine in which leaf to complete the search. We can repeat this procedure to handle arbitrarily large sets of ranges and next-hop indices. For each level added, the number of ranges that can be handled increases by a factor of 17. Hence, by using t levels, we can handle

$$DLT\_num\,(t) = \begin{cases} 8 & \text{, for } t = 1 \\ 17 \cdot DLT\_num\,(t-1) & \text{, otherwise} \end{cases}.$$

For $t = 1, 2, 3$ we get the sizes 8, 136 and 2312 for *complete trees of height* 1, 2, and 3 trees respectively. When $n \geq 202$ we can afford to use partially filled cache lines at all levels at the same time. In some cases, we will however store a leaf directly below a level 3 node. When arriving at that leaf, we need to a way to tell if it is a node or a leaf. This is achieved by storing the first two range boundaries in decreasing order in a leaf (but not in a node), swapping them before searching, and swapping them back after completing the search. The extra cost for this encoding/decoding is negligible. The size of a t level complete tree, measured in number of cache lines, is given by

$$DLT\_size\,(t) = \begin{cases} 1 & \text{, for } t = 1 \\ 1 + 17 \cdot DLT\_size\,(t-1) & \text{, otherwise} \end{cases}.$$

For $n \leq 201$, the cost for incomplete trees resulting in partially filled cache lines becomes too high with respect to the number of ranges handled. For example, when n is in the range $137 \ldots 201$ only one single 16 bits range boundary is used in the level 3 node. To reduce quantization effects we will try to store partially filled nodes and leafs as compact as possible without introducing extra memory accesses for the lookup. Let

$$\begin{aligned} k_2 &= \left\lfloor \frac{n}{136} \right\rfloor \\ n_2 &= n \bmod 136 \\ k_1 &= \left\lfloor \frac{n_2}{8} \right\rfloor \\ n_1 &= n_2 \bmod 8 \end{aligned}$$

For $30 \leq n \leq 201$, k_2 is the number of complete level 2 trees and n_2 is the number of remaining ranges after storing as many as possible in complete level 2 trees. Similarly, k_1 is the number of complete level 1 trees and n_1 is the number of ranges remaining after storing as many as possible in complete level 2 and level 1 trees. If $n > 136$, we need a partial level 3 node consisting of exactly one range boundary. In addition, we need a partial level 2 node if $n \neq 136$ and finally, we need a partial level 1 leaf if $n_1 \neq 0$. The number of range boundaries we need to store in the partial level 2 node equals $k_1 - 1$ if $n_1 = 0$ and k_1 otherwise. By accounting for storing high 2 bits of next-hop indices, range boundaries, and low16 bits of next-hop indices, the number of 16 bits blocks required to store the partial leaf becomes

$$\begin{aligned} u_1(n) &= 1 + n_1 + (n_1 - 1) \\ &= 2n_1 \\ &= 2 \cdot (n_2 \bmod 8) \\ &= 2 \cdot ((n \bmod 136) \bmod 8) \end{aligned}$$

The total number of 16 bit blocks required to store the partial nodes and leaf is then given by

$$u(n) = u_1(n) + u_2(n) + u_3(n),$$

where $u_1(n)$ s defined as above,

$$\begin{aligned} u_2(n) &= \begin{cases} k_1 - 1 & \text{, if } n_1 = 0 \\ k_1 & \text{, otherwise} \end{cases} \\ &= \begin{cases} \left\lfloor \frac{n_2}{8} \right\rfloor - 1 & \text{, if } n_2 \bmod 8 = 0 \\ \left\lfloor \frac{n_2}{8} \right\rfloor & \text{, otherwise} \end{cases} \\ &= \begin{cases} \left\lfloor \frac{n \bmod 136}{8} \right\rfloor - 1 & \text{, if } (n \bmod 136) \bmod 8 = 0 \\ \left\lfloor \frac{n \bmod 136}{8} \right\rfloor & \text{, otherwise} \end{cases} , \end{aligned}$$

and

$$u_3(n) = \begin{cases} 0 & \text{, if } n \leq 136 \\ 1 & \text{, otherwise} \end{cases} .$$

We call the partial leaf and nodes the *head* of the tree and the remaining complete level 2 tree and level 1 nodes the *tail* of the tree. The tail consists of 32 byte blocks and is stored in a memory area designated for 32 byte aligned structures. The head, on the other hand, consists of sufficiently many 8 byte block for storing the $u(n)$ 16 bits blocks and a 32 bits pointer to the tail. As shown in Figure 5, which shows a head with $u_1(n) = 12$, $u_2(n) = 4$, and $u_3(n) = 1$, the tail pointer is stored in the first 8 bytes block together with the level 3 node (L3). Since each 8 byte block is completely contained within a cache line, at most 1 memory access is required for searching the head at level 3, leaving 2 memory accesses as required for completing the search at level 2 in the tail. The maximum value of $u_2(n)$ is 16 (complete level 2 node). This means that the level 2 note stored in the head can not be distributed over more than 2 cache lines. The first part of the node will be stored in the same 8 byte block (and therefore also the same cache line) as the tail pointer, which means that the first memory access is already accounted for. In the second memory access we are guaranteed to complete the level 2 search in the head, leaving the third and last memory access for completing the search in a leaf stored either in the head or in the tail.

At level 3 and 4 range sub-sets from the same sub-universe $\mathbf{U}_i$ (first partitioning step) can share Dynamic Layered Tree as long as the number of memory accesses for completing the lookups does not exceed 2 and 1 respectively. For example, consider the three range sets $\mathbf{R}_{i,j_1}$, $\mathbf{R}_{i,j_2}$, and $\mathbf{R}_{i,j_3,k}$ where $j_1 < j_2 < j_3$, $|\mathbf{R}_{i,j_1}| = 87$, $|\mathbf{R}_{i,j_2}| = 44$, and $|\mathbf{R}_{i,j_3,k}| = 5$. These can be represented using a complete level 2 DLT since the total number of ranges is $87 + 44 + 5 = 136$ and the choice of j_1, j_2, and j_3 guarantees that the representation of $\mathbf{R}_{i,j_3,k}$ does not straddle a cache line boundary. The pointers referring to the sub-structures representing $\mathbf{R}_{i,j_1}$ and $\mathbf{R}_{i,j_2}$ will refer to the complete level 2 tree and the pointer referring to the sub-structure representing $\mathbf{R}_{i,j_3,k}$ will refer to the leaf of the complete 2 tree containing $\mathbf{R}_{i,j_3,k}$.

5.2.5 Static Flat Trees

A *Static Flat Tree* is used at level 4 to represent a sorted list of 4 bit range boundaries $r_1, r_2, \ldots, r_{n-1}$ and the corresponding list of next-hop indices $d_1, d_2, \ldots, d_n$. It is used only when n equals 9, 10, or 11. As above, d_1 is associated with $0..r_1$, d_2 is associated with $r_1 + 1..r_2$, and so on until finally d_n is associated with $r_{n-1} + 1..\infty$. The data structure and the search method is basically the same as a dynamic layered tree leaf but with one difference. Instead of storing 7 16 bits range boundaries and 8 next-hop indices, we store 10 4 bits range boundaries and 11 next-hop indices (we fill the tree up to 11 ranges even if we have 9). The total number of bits required for this is $10 \cdot 4 + 11 \cdot 18 = 238$. Figure 6 illustrates a Static Flat Tree containing 11 ranges with 8 unused bits in the R-area and 10 in the H-area.

5.3 Compressed Forwarding Table Lookup

Up to this point, we have described how to convert the prefix matching problem into a range matching problem. We have also described all the pieces necessary for achieving a space efficient representation of the set of ranges - a representation in which we can perform the range matching operation in four memory accesses, thereby achieving an efficient forwarding table representation.

In this section, we describe the lookup procedures in more detail.

5.3.1 Main Lookup Procedure

The main lookup function is represented by the static vector node lookup *SVN_lookup*. It accepts two parameters, the IPv4 address a and a pointer to the data structure

T. Initially, T refers to the level 1 SVN containing $T_0, \ldots, T_{2^{16}-1}$ (see Figure 1). The array elements are all 32 bits non-negative integers so indexing and retrieving the sub-level pointer is straight forward. In line $1 \ldots 12$ the first level lookup is performed. Depending on the result from encoding the pointer value, the lookup is either completed (line 3) or continues by calling a custom lookup procedure (lines 6, 8, or 10). If neither of these applies, the lookup continues with the next level SVN (line 13).

```
$SVN\_lookup(T, a)$
    $T \leftarrow T_{a[31...16]}$
    if $T[31...24] = 0$ then
        return $T[23...0]$
    elsif $T[31...24] \neq 1$ then
        if $T[31...24] \leq 29$ then
            return $DFT\_lookup(T[23...0], T[31...24], a[15...0])$
        elsif $T[31...24] \leq 201$ then
            return $DLT\_lookup^1(T[23...0], T[31...24], a[15...0])$
        else
            return $DLT\_lookup^2(T[23...0], a[15...0])$
        end
    end
    $T \leftarrow T[23...0]$
    $T \leftarrow T_{a[15...8]}$
    if $T[19...18] = 0$ then
        return $T[17...0]$
    elsif $T[19...18] \neq 1$ then
        return $DLT\_lookup^3(T[17...0], a[15...0])$
    end
    $T \leftarrow T[17...0]$
    $T \leftarrow T_{a[7...4]}$
    if $T[19...18] = 0$ then
        return $T[17...0]$
    elsif $T[19...18] \neq 1$ then
        if $T[19...18] = 2$ then
            return $DLT\_lookup^3(T[17...0], a[15...0])$
        else
            return $SFT\_lookup(T[17...0], a[15...0])$
        end
    end
    $T \leftarrow T[17...0]$
    return $T_{a[3...0]}$
```

The second level SVN lookup begins in line 14 by indexing into the array represented by T which contains pointers $T_0, \ldots, T_{2^8-1}$. Level 2 nodes are packaged in groups of three. The 16 most significant bits of the pointer are used to address the group and the 2 least significant bits are used to locate the node within the group. Within the group, the first four pointers in each cache line belongs to the first node, the next four pointers in each cache line belongs to the second node, and the last four pointers within each cache line belongs to the third node. The high bits from the address $a\,[15 \ldots 10]$ are used to locate the cache line, the 2 least significant bits from the node pointer are used to locate the group of four pointers within the cache line, and the low bits from the address $a\,[9 \ldots 8]$ are use to index within the group of pointers. Depending on the result from encoding the retrieved pointer, the lookup is either finished (line 16), continued by calling the custom procedure (line 18), or continued with the next SVN level (line 20).

When entering the third level of the lookup, T refers to an array containing pointers $T_0, \ldots, T_{2^4-1}$. Also level 3 nodes are packaged in groups of three and the pointers are interpreted in the same way as at level 2 to retrieve the location of the group and the node within the group. The indexing is basically the same. We use the high bits from the address $a\,[7 \ldots 6]$ to locate the cache line, and the 2 least significant bits from the node pointer to locate the group of four pointers within the cache line, and the low bits from the address $a\,[5 \ldots 4]$ are use to index within the group of pointers. After decoding the pointer, we know whether the lookup is finished (line 23) or whether to continue the lookup in a custom procedure (line 26 or 28). Otherwise, the lookup is continued at the next and final level -- level 4.

At level 4, T refers to an array containing next-hops $T_0, \ldots, T_{2^4-1}$. The nodes are packaged together in groups of 7. Therefore, the 15 most significant bits of the node pointer are used to locate the group and the 3 least significant bits are used to locate the node within the group. We use the same kind of organization within each cache line as above. That is, the first two (instead of four) pointers belongs to the first group, the next two pointers belongs to the second group and so on until the last two pointers that belongs to the seventh group. Consequently, we must use the high address bits $a\,[3 \ldots 1]$ to locate the cache line, the lowest three pointer bits to locate the group if two next-hop indices within the cache line, and the lowest address bit $a\,[0]$ to index within the group to retrieve the next-hop index.

5.3.2 Index Search

The key operation in all custom lookup procedures (except *SVN_lookup*) is an efficient binary search that is used to compute indices into arrays of next-hop indices. Given a array of range boundaries $r_0, \ldots, r_{n-1}$, sorted in increasing order, and an address a we compute the minimum index i satisfying $a \leq r_i$. If no such r_i is present, the result is n. Throughout the procedure, we heavily exploit pointer arithmetics, combined with interpretation of boolean values as numerical values, to achieve a procedure with a minimum of conditional branches. That is, for a pointer r representing the address to the first element of the array we can write r_i as $(r+i)_0$. Moreover, the boolean values false and true will be used as 0 and 1 respectively in computations.

```
ix_search (r, n, a)
    $b \leftarrow r$                                    1
    $k \leftarrow \lfloor \lg n \rfloor$                2
    $m \leftarrow 2^k$                                  3
    $r \leftarrow r + (n - m) \cdot (r_{m-1} < a)$      4
    while $k > 0$ do                                    5
        $k \leftarrow k - 1$                            6
        $r \leftarrow r + (r_{2^k} < a) \cdot 2^k$      7
    end                                                 8
    return $(r - b) + (r_0 < a)$                        9
```

In lines 1 to 3 we record the address of the first element in b, compute floor of the base 2 logarithm of n and assigns it to k, and compute 2 to the power of k and assigns the value to m. The idea is to quickly determine if the search is to be performed among the m first range boundaries or among the m last range boundaries. When this is done, we can repeatedly compare the middle element and cut an array of size 2^k in half (decreasing k in each step) until only one element remains. The actual decision and possible modification of r is performed in line 3. If $r_{m-1} \geq a$ the numerical value of $(r_{m-1} < a)$ is 0 and r is assigned to itself (no modification). Otherwise, the start address of r is moved $n - m$ steps forward. In either case, the remaining search is performed among $m = 2^k$ elements.

The query key a is repeatedly compared to the middle element and r is modified to reflect the result of the comparisons until only one element remains (lines 5...8).

Finally, the difference between the location of the first element in the original array and the element remaining is computed and after adding the result from the comparison with the last element the result is returned (line 9).

5.3.3 Dynamic Flat Tree Lookup

The dynamic flat tree structure is stored in an array of 16 bits memory slots

$$T_1, T_2, \ldots, T_{\lceil \frac{n}{8} \rceil + 2n - 1}.$$

The actual organization of the tree depends on the size and the location T of the first memory slot with respect to cache line boundaries. If $n \leq 16$ or the first memory slot is located at the beginning of a cache line the tree is organized as follows: $H = h_1, \ldots, h_n$ is stored in $T_{1 \ldots \lceil \frac{n}{8} \rceil}$; $r_1, \ldots, r_{n-1}$ are stored in $T_{\lceil \frac{n}{8} \rceil + 1 \ldots \lceil \frac{n}{8} \rceil + n - 1}$, and $l_1, \ldots, l_n$ are stored in $T_{\lceil \frac{n}{8} \rceil + n \ldots \lceil \frac{n}{8} \rceil + 2n - 1}$. The procedure begins by computing the cache line boundary offset q (line 1). Depending on the value of n and q ($q = 0$ implies that the first memory slot is located at the beginning of a cache line), the lookup is either completed in line 3 and 4 or continues with the more complex cases in line 6. In line 3 the index $(0 \ldots n-1)$ of the next-hop index is computed using *ix_search*, and in line 4 the resulting next-hop index is assembled (and returned) by appending $h_{i+1} = T_{\lfloor \frac{i}{8} \rfloor} [i \bmod 8 \ldots (i \bmod 8) + 1]$ to $l_{i+1} = T_{\lceil \frac{n}{8} \rceil + n + i}$.

```
$DFT\_lookup(T, n, a)$
    $q \leftarrow (16 - (T \bmod 16)) \bmod 16$                                                    1
    if $(n \leq 16) \vee (q = 0)$ then                                                              2
        $i \leftarrow ix\_search\left(\left\langle T_{\lceil \frac{n}{8} \rceil + 1}, \ldots, T_{\lceil \frac{n}{8} \rceil + n - 1} \right\rangle, a\right)$      3
        return $T_{\lfloor \frac{i}{8} \rfloor} [i \bmod 8 \ldots (i \bmod 8) + 1] \, T_{\lceil \frac{n}{8} \rceil + n + i}$      4
    end                                                                                             5
    $i \leftarrow ix\_search\left(\left\langle T_{q + \lceil \frac{n}{8} \rceil + 1}, \ldots, T_{q + \lceil \frac{n}{8} \rceil + n - 1} \right\rangle, a\right)$      6
    if $i < n - q$ then                                                                             6
        return $T_{\lfloor \frac{i}{8} \rfloor} [i \bmod 8 \ldots (i \bmod 8) + 1] \, T_{\lceil \frac{n}{8} \rceil + n + i}$      7
    end                                                                                             8
    return $T_{\lfloor \frac{i}{8} \rfloor} [i \bmod 8 \ldots (i \bmod 8) + 1] \, T_{i + q - n + 1}$      9
```

In the more complex cases ($n > 16$ and $q \neq 0$) the tree is organized as follows: $H = h_1, \ldots, h_n$ is stored in $T_{q+1 \ldots q+\lceil \frac{n}{8} \rceil}$, $r_1, \ldots, r_{n-1}$ are stored in $T_{q+\lceil \frac{n}{8} \rceil+1 \ldots q+\lceil \frac{n}{8} \rceil+n-1}$, $l_1, \ldots, l_{n-q}$ are stored in $T_{q+\lceil \frac{n}{8} \rceil+n \ldots 2n-1+\lceil \frac{n}{8} \rceil}$, and $l_{n-q+1}, \ldots, l_n$ are stored in $T_{1 \ldots q}$. The computation of the index (line 6) is essentially the same as above except for considering the cache line boundary offset. If the low 16 bits from the resulting next-hop index are located at the end of the occupied memory area ($i < n-q$), the retrieval and assembly of the pieces are straight forward (line 7). Otherwise, the low 16 bits from the resulting next-hop index are located at the beginning of the memory area. Retrieval and assembly is finally performed in line 9.

5.3.4 Dynamic Layered Tree Lookup

We distinguish between two major kinds of dynamic layered tree lookup. The straight forward version *DLT_lookup*^{2,3} (two minor kinds within this major) is used either when the trees are complete or when the size is large enough to allow us to disregard quantization effects. When the trees are small, it is necessary to pay attention to quantization effects. As described in Section 5.2.4, this requires a more complex representation and the need for a corresponding lookup procedure – *DLT_lookup*^{1}.

The *DLT_lookup*^{1} procedure accepts three arguments: T, n, and a, where T represents the *head* of the tree structure stored in an array of 16 bits memory slots $T_1, T_2, \ldots$, and n and a are the size and query address (least significant 16 bits of the original address) respectively. In line 1 we extract the pointer to the *tail* of the tree and assigns the value to t for later use. Since the starting point of the tail is 32 byte aligned, t is actually an index to a cache line. This is followed by initializing an offset variable o to 0 (line 2). If $\left\lceil \frac{n}{136} \right\rceil > 1$ as tested in line 3, the head contains a partial level 3 (L3) node. Since $n \leq 201 < 2 \cdot 136$, we know that the partial L3 node contains exactly one range boundary stored in T_3. If $a \leq T_3$ the straight forward lookup procedure is called in line 5 to finish the lookup in *the* complete L2 sub-tree stored in the beginning of the tail and referred to by t. Otherwise, the complete L2 tree is skipped by adding its size to t (line 7) and the offset o is increased by the size (i.e. 1) of the partial L3 node (line 8). In line 10 the size n_2 (number of range boundaries) of the partial L2 node is computed, followed by the index search (line 11).

```
$DLT\_lookup^1\,(T,\,n,\,a)$
    $t \leftarrow T_{1\ldots2}$                                                   1
    $o \leftarrow 0$                                                          2
    if $\left\lceil \frac{n}{136} \right\rceil > 1$ then                                              3
        if $a \le T_3$ then                                                   4
            return $DLT\_lookup^{2,3}\,(t, 2,\, a)$                              5
        end                                                           6
        $t \leftarrow t + 1 + 17$                                               7
        $o \leftarrow o + 1$                                                    8
    end                                                               9
    $n_2 \leftarrow \left\lceil \frac{n \bmod 136}{8} \right\rceil - 1$                                       10
    $i \leftarrow ix\_search\,(\langle T_{2+o+1}, \ldots, T_{2+o+n_2} \rangle\,, n_2, a)$              11
    if $i < n_2$ then                                                    12
        return $DLT\_lookup^{2,3}\,(t + i, 1,\, a)$                              13
    end                                                              14
    $n_1 \leftarrow (n \bmod 136) \bmod 8$                                        15
    $i \leftarrow ix\_search\,(\langle T_{2+o+n_2+1}, \ldots, T_{1+o+n_2+n_1} \rangle\,, n_1 - 1, a)$      16
    return $T_{2+o+n_2+2n_1}\,[i \bmod 8 \ldots (i \bmod 8) + 1]\, T_{2+o+n_2+n_1+i}$       17
```

If $i < n_2$ the lookup continues by searching in the $(i+1)$th complete L1 tree stored in the tail at $t+i$ (line 13). Otherwise, the lookup is completed by searching the partial L1 tree (leaf) stored in the head. This is accomplished as follows. First the size n_1 of the partial L1 tree is computed (line 15). The range boundaries $r_1, \ldots, r_{n_1-1}$ are stored in

$$T_{2+o+n_2+1}, \ldots, T_{1+o+n_2+n_1};$$

the low bits of the next-hop indices $l_1, \ldots, l_{n_1}$ are stored in

$$T_{1+o+n_2+n_1+1}, \ldots, T_{1+o+n_2+2n_1},$$

and the high bits of the next-hop indices $h_1, \ldots, h_{k_1}$ are stored in $T_{2+o+n_2+2n_1}$. It remains to perform the index search (line 16) and extract and assemble the next-hop index (line 17).

The straight forward lookup $DLT\_lookup^{2,3}$ (below) is slightly simpler. It accepts three arguments: T which refers to a array of 16 bits memory slots $T_1, \ldots, T_{16}$, the level

t, and the query key a. Independent on the value of t, the tree referred to by T might be either a node or a leaf. Nevertheless, there are 7 range boundaries $r_1, \ldots, r_7$ stored in $T_1, \ldots, T_7$. A node it contains 16 range boundaries $r_1, \ldots, r_{16}$ stored in $T_1, \ldots, T_{16}$. The first two range boundaries of a node are stored in sorted order. This is tested in line 1 followed by performing an index search. Thereafter, the $DLT\_lookup^{2,3}$ is recursively called to search the next level sub-tree stored at $T + 1 + i \cdot DLT\_size\,(t)$ (see line 2...3 and the definition of DLT_size in Section 5.2.4).

```
$DLT\_lookup^{2,3}\,(T, t, a)$
    if $T_1 < T_2$ then                                                        1
        $i \leftarrow ix\_search\,(\langle T_1, \ldots, T_{16}\rangle, 16, a)$                       2
        return $DLT\_lookup^{2,3}\,(T + 1 + i \cdot DLT\_size\,(t), t - 1, a)$       3
    end                                                                        4
    $i \leftarrow ix\_search\left(\underbrace{\langle T_2, T_1, T_3 \ldots, T_7\rangle}_{T_1 \text{ and } T_2 \text{ are swapped}}, 7, a\right)$         5
    return $T_{16}\,[i \bmod 8 \ldots (i \bmod 8) + 1]\,T_{8+i}$                         6
```

A leaf, it is organized as follows: the range boundaries $r_1, \ldots, r_7$ stored in $T_1, \ldots, T_7$, low bits from next-hop indices $l_1, \ldots, l_8$ stored in $T_8, \ldots, T_{15}$, and the high bits from next-hop indices $h_1, \ldots, h_8$ are stored in T_{16}. Searching the leaf is simply accomplished by first swapping the two first range boundaries, performing an index search to compute i and then swapping the first two range boundaries back (line 5). The last step is to assemble the pieces from the next-hop index and return the result.

5.3.5 Static Flat Tree Lookup

A Static Flat Tree (SFT) is very similar to a dynamic layered tree leaf. The first difference is that the size of the range boundaries is 4 bits in the SFT compared to 16 in the DLT. This requires a special version ix_search^4 for searching the 4 bits range boundaries. The second difference is the number of range boundaries which is 10 in the SFT compared to 7 in the DLT. The tree consists of an array of 16 bits memory slots $T_1, \ldots, T_{16}$ organized as follows: the 4 bit range boundaries $r_1, \ldots, r_{10}$are stored in $T_{1\ldots3}$, the low 16 bits of the next-hop indices $ll_1, \ldots, l_{11}$ are stored in $T_4, \ldots, T_{14}$, and the high 2 bits of the next-hop indices are stored in $T_{15\ldots16}$.

```
$SFT\_lookup\,(T,\ n,\ a)$
        $i \leftarrow ix\_search^4\,(\langle T_1\,[3\ldots0]\,,\ldots,T_3\,[7\ldots4]\rangle\,,10,a)$    1
        return $T_{\lfloor \frac{i}{8} \rfloor}\,[i \bmod 8\ldots(i \bmod 8)+1]\,T_{4+i}$    2
```

As in DLT leafs, lookup is achieved by performing an index search (line 1). The complete array of 4 bits range boundaries is given by

$$\langle T_1\,[3\ldots0]\,,T_1\,[7\ldots4]\,,T_1\,[11\ldots8]\,,T_1\,[15\ldots12]\,,T_2\,[3\ldots0]\,,$$
$$T_2\,[7\ldots4]\,,T_2\,[11\ldots8]\,,T_2\,[15\ldots12]\,,T_3\,[3\ldots0]\,,T_3\,[7\ldots4]\rangle\,.$$

This is followed by accessing and assembling the next-hop index and returning the result (line 2).

5.4 Router System Architecture

Figure 7 is a schematic block diagram of a router system architecture chosen as an example. There are three autonomous processes: the *Routing Process*, the *Builder Process*, and the *Forwarding Process*, and three data structures: the Routing Table, the Forwarding Table, and the Next-hop Table.

All datagrams entering and exiting the system are handled by the Forwarding Process. The basic operation is to receive a packet (fp1), lookup the next-hop index in the forwarding table using $SVN\_lookup\,(T,\ a)$ where T is the forwarding table and a is the destination address (fp2), retrieve the next-hop information from the next-hop table (fp3), and forward the packet according to the next-hop information (fp4).

The Routing Process, or *Routing Protocol Daemon*, is a process that communicates with other systems in the router's neighborhood using a *routing protocol* to learn about changes in network topology and to update the routing table to accommodate for such changes. An example of a routing process, or more precisely a computer program product defining the routing process, is the *Gate Daemon* (gateD) available from www.gated.org. Different routing protocols works in different ways and use different optimization criteria to determine which is the most efficient path through the network and thereby to determine what updates to perform on the routing table. The basic operation of all router processes is however to detect or learn about network topology changes by communicating according to the routing protocol (rp1), update the routing table to accommodate for these changes (rp2), and inform the neighbors (routing processes) about topology changes (rp3).

As opposed to the *dynamic* routing table, the *semi-static* forwarding table is not designed to be directly manipulated by the routing process. Dynamic and static here refers to the computational complexity of performing table updates – dynamic means *easy* and static means that the complete table must be rebuild. Modifications of the routing table are propagated into modifications of the forwarding table in a controlled manner by the workings of a Builder Process. There are several ways of scheduling forwarding table rebuilds or partial rebuilds and a commonly used method is to slightly postpone the next rebuild after detecting a routing table update. For example, the builder process can accumulate all changes to the routing table during a pre-determined period of time before executing the rebuild. In this way, a burst of routing table modifications (which is not uncommon) can be handled by a single forwarding table rebuild (or update). The basic operation of the Builder Process is to wait until a routing table update occurs (bp1), retrieve the range boundaries and next-hop indices using the generic method described in Section 5.1.2 (bp2), or a more efficient method optimized for the routing table representation used, and rebuild the forwarding table according to the representation described in Section 5.2 (bp3).

5.5 Small Routing and Forwarding Tables

A simple and straight forward way of representing the routing table is to store address prefixes together with next-hop indices (NH) in entries that are linked to each other in a layered tree-like structure as shown in Figure 7. Each entry has a pointer to the *next* entry to the right (if present) and to the *first* child entry (if present). Entries at the same level are stored in sorted order with respect to the smallest address matching the prefix of the entry (i.e. the starting point of the range constituted by the prefix). Entries with prefixes that are sub-sets of the prefix of a given entry are stored in levels below the given entry. The detailed information stored in the routing table entries shown in Figure 7 is shown in the table below.

i	p_i (binary representation)	p^{lo}	p^{hi}	NH
1	*	0	4294967295	1
2	0000*	0	268435455	9
3	00000000000000000000001010011010	666	666	16
4	0000001100*	50331648	54525951	7
5	001000100101101000011100001*	576330784	576330815	15
6	011111011010100111*	2108276736	2108293119	12
7	1*	2147483648	4294967295	4
8	100*	2147483648	2684354559	10
9	101001101011101100*	2797273088	2797289471	3
10	101101111101000001*	3083878400	3083894783	5
11	1100010001100110 1*	3295051776	3295084543	14
12	1100100001110100110111010*	3363101952	3363102079	6
13	1111000101100100*	4049862656	4049928191	2
14	11110010111000010*	4074831872	4074864639	13
15	1111010111111001011010111111*	4126764016	4126764031	8
16	111111011011001101101*	4256393216	4256395263	11

In this example, the result from traversing the routing table and compute the intermediate range partition representation results in the following.

i	r_i^{lo}	r_i^{hi}	Next-hop Index
1	0	665	9
2	666	666	16
3	667	50331647	9
4	50331648	54525951	7
5	54525952	268435455	9
6	268435456	576330783	1
7	576330784	576330815	15
8	576330816	2108276735	1
9	2108276736	2108293119	12
10	2108293120	2147483647	1
11	2147483648	2684354559	10
12	2684354560	2797273087	4
13	2797273088	2797289471	3
14	2797289472	3083878399	4
15	3083878400	3083894783	5
16	3083894784	3295051775	4
17	3295051776	3295084543	14
18	3295084544	3363101951	4
19	3363101952	3363102079	6
20	3363102080	4049862655	4
21	4049862656	4049928191	2
22	4049928192	4074831871	4
23	4074831872	4074864639	13
24	4074864640	4126764015	4
25	4126764016	4126764031	8
26	4126764032	4256393215	4
27	4256393216	4256395263	11
28	4256395264	4294967295	4

To represent this in a forwarding table, we use a special kind of Dynamic Layered Tree suitable for smaller routing tables. The principle is exactly the same as for regular DLTs but instead of using 16 bits range boundaries, we use 32 bits range boundaries. Moreover, the number of bits used for representing next-hop indices is reduced to 16.

In this data structure which we call 32-bit Dynamic Layered Tree a node consists of 8 range boundaries instead of 16 and a leaf contains 5 range boundaries and 6 next-hop indices instead of 7 and 8 respectively. This means that we can represent up to $6 * 9 * 9 * 9 = 4374$ ranges or 2187 routes using this technique without exceeding the design limit of 4 memory accesses. In this configuration, the maximum amount of memory required to represent 2187 routes is

$$32 + 9 \cdot (32 + 9 \cdot (32 + 9 \cdot 32)) = 26240 \text{ bytes.}$$

The 32-bit DLT build representing the range partition above is shown in Figure 8. Each row represents a cache line that stores either a node (the first three rows) or a leaf (the last five rows). Observe that quantization effects are disregarded which means that this particular representation is slightly less memory space efficient than the full flown compressed forwarding table for large routing tables. The customized lookup and index search procedures for 32-bit DLTs are shown in C-code below.

```
static __inline int
DLT32_ix_search5(u_int32_t *tkey, u_int32_t qkey)
{
    u_int32_t *base = tkey;
    tkey += *tkey <= qkey;
    tkey += (*(tkey + 2) <= qkey) << 1;
    tkey += (*(tkey + 1) <= qkey) << 0;
    return (int)((tkey - base) + (*tkey <= qkey));
}

static __inline int
DLT32_ix_search8(u_int32_t *tkey, u_int32_t qkey)
{
    u_int32_t *base = tkey;
    tkey += (*(tkey + 4) <= qkey) << 2;
    tkey += (*(tkey + 2) <= qkey) << 1;
    tkey += (*(tkey + 1) <= qkey) << 0;
    return (int)((tkey - base) + (*tkey <= qkey));
}

#define _FT_SIZE_1 (8                    )
#define _FT_SIZE_2 (8 + 9 * _FT_SIZE_1)
#define _FT_SIZE_3 (8 + 9 * _FT_SIZE_2)

static __inline int
DLT32_lookup(u_int32_t *ft, u_int32_t key)
{
    ft += 8 + _FT_SIZE_3 * _ft_qsearch8(ft, key);
    ft += 8 + _FT_SIZE_2 * _ft_qsearch8(ft, key);
    ft += 8 + _FT_SIZE_1 * _ft_qsearch8(ft, key);
    return ((u_int16_t *)(ft + 5))[_ft_qsearch5(ft, key)];
}
```

`DLT32_ix_search5` and `DLT32_ix_search8` are custom index search procedures for searching among 5 and 8 range boundaries respectively. Since both the number of levels (four) and the number of range boundaries to search is known in each step the lookup can be performed as a deterministic sequence of arithmetical operations, without conditional branching, to fully exploit the pipelining of instructions performed by the CPU.

6 Claims

1. A method for IP routing lookup to determine where to forward an IP-datagram with a given destination address by retrieving from a routing table a next-hop index indicating where to forward said datagram, said next-hop index being associated with the longest matching prefix of said destination address, said address being a number in an address universe **U**, wherein a set of address prefixes **P** and a mapping of **P** onto a set of next-hop indices D are converted into a set of ranges **R**, constituting a partition of **U**, and a mapping of **R** onto D, **characterized** by the steps of building and storing in a memory a forwarding table representation from **R** and D by using a predetermined layered data structure where the construction of the layer is selected depending on the range density $|\mathbf{R}'|$ for the sub-universe $\mathbf{U}'$ represented by that layer to get a space efficient representation of the set of ranges **R**, and performing the lookup by a range matching operation in said forwarding table.

2. The method according to claim 1, **characterized** in that the number of routes to be represented in the forwarding table and a maximum allowable number of memory accesses for the lookup are prescribed, whereupon said data structures are selected as construction blocks for the building of the forwarding table in such a way that memory needs are minimized.

3. The method according to claim 1, **characterized** in that the number of routes to be represented in the forwarding table and a maximum available memory capacity are prescribed, whereupon said data structures are selected as construction blocks for the building of the forwarding table in such a way that the number of memory accesses for the lookup are minimized.

4. The method according to any of the preceding claims, **characterized** in that said data structures for the building of the forwarding table are selectable among the structures Dynamic Flat Tree, Dynamic Layered Tree and Static Flat Tree and variations thereof.

5. The method according to any of the preceding claims, **characterized** in that to repeatedly reduce the size of the address sub-universes in successive levels

a Static Vector Node (SVN) representation is used in a first level for the forwarding table representation, one of a first set of said predetermined layered data structures, selectable depending on the actual value of the range density $|\mathbf{R}_i|$ to give the most efficient table representation, is chosen for building the forwarding table in a second level, if $|\mathbf{R}_i|$ is less than a predetermined first limit value, otherwise a SVN representation is used, in a third level one of a second set of said predetermined data structures, selectable depending on the actual value of the range density $|\mathbf{R}_{i,j}|$ to give the most efficient table representation, is chosen for building the forwarding table in this third level, if $|\mathbf{R}_{i,j}|$ is less than a predetermined second limit value, otherwise a SVN representation is used, etc. until all levels have been represented.

6. The method according to claim 5, **characterized** in that an initial SVN lookup by direct indexing is performed in said first level to retrieve a pointer that either represents a next-hop index, which means that the lookup is completed, or refers to a sub-data structure where the lookup is continued on said second level and, depending on the value of $|\mathbf{R}_i|$, encoded in the pointer value and obtained by decoding the pointer value, a lookup is completed in a forwarding table represented by one of said first set of said predetermined layered data structures if $|\mathbf{R}_i|$ is less than said predetermined first limit value, otherwise a SVN lookup by direct indexing is performed in said second level to retrieve a pointer that either represents a next-hop index, which means that the lookup is completed, or refers to a sub-data structure where the lookup is continued on said third level and, depending on the value of $|\mathbf{R}_{i,j}|$, encoded in the pointer value and obtained by decoding the pointer value, a lookup is completed in a forwarding table represented by one of said second set of said predetermined layered data structures if $|\mathbf{R}_{i,j}|$ is less than said predetermined second limit value, otherwise a SVN lookup by direct indexing is performed etc. until the lookup is completed.

7. The method according to any of the claims 1 – 2 or 4 – 6 **characterized** in that the maximum number of routes is fixed to 2^{18} routes and the maximum allowable number of memory accesses to 4, whereupon said data structures are selected as construction blocks for the building of the forwarding table in such

a way that memory needs are less than 2.7 Mbytes.

8. The method according to any of the claims 1 or 3 – 6, **characterized** in that the maximum number of routes is fixed to 2^{18} routes and the maximum available memory capacity to 2.7 Mbytes, whereupon said data structures are selected as construction blocks for the building of the forwarding table in such a way that the number of memory accesses for the lookup is less than or equal to 4.

9. The method according to any of the preceding claims, **characterized** in that to repeatedly reduce the size of the address sub-universes in successive levels a Static Vector Node (SVN) representation of size 2^{16} pointers is used in a first level for the forwarding table representation, one of a first set of said predetermined layered data structures, selectable depending on the actual value of the range density $|\mathbf{R}_i|$ to give the most efficient table representation, is chosen for building the forwarding table in a second level, if $|\mathbf{R}_i|$ is less than a predetermined first limit value 2313, otherwise a SVN representation of size 2^8 pointers is used, in a third level one of a second set of said predetermined data structures, selectable depending on the actual value of the range density $|\mathbf{R}_{i,j}|$ to give the most efficient table representation, is chosen for building the forwarding table in this third level, if $|\mathbf{R}_{i,j}|$ is less than a predetermined second limit value 137, otherwise a SVN representation of size 2^4 pointers is used, and in a fourth level one of a third set of said predetermined data structures, selectable depending on the actual value of the range density $|\mathbf{R}_{i,j,k}|$ to give the most efficient table representation, is chosen for building the forwarding table in this forth level, if $|\mathbf{R}_{i,j,k}|$ is less than a predetermined third limit value 12, otherwise a SVN representation of size 2^4 pointers is used.

10. The method according to claim 9, **characterized** in that an initial SVN lookup by direct indexing is performed in said first level to retrieve a pointer that either represents a next-hop index, which means that the lookup is completed, or refers to a sub-data structure where the lookup is continued on said second level and, depending on the value of $|\mathbf{R}_i|$, encoded in the pointer value and obtained by decoding the pointer value, a lookup is completed in a forwarding table represented by one of said first set of said predetermined layered data structures if $|\mathbf{R}_i|$ is less than said predetermined first limit value 2313, otherwise

a SVN lookup by direct indexing is performed in said second level to retrieve a pointer that either represents a next-hop index, which means that the lookup is completed, or refers to a sub-data structure where the lookup is continued on said third level and, depending on the value of $|\mathbf{R}_{i,j}|$, encoded in the pointer value and obtained by decoding the pointer value, a lookup is completed in a forwarding table represented by one of said second set of said predetermined layered data structures if $|\mathbf{R}_{i,j}|$ is less than said predetermined second limit value 137, otherwise a SVN lookup by direct indexing is performed in said third level to retrieve a pointer that either represents a next-hop index, which means that the lookup is completed, or refers to a sub-data structure where the lookup is continued on said fourth level and, depending on the value of $|\mathbf{R}_{i,j,k}|$, encoded in the pointer value and obtained by decoding the pointer value, a lookup is completed in a forwarding table represented by one of said third set of said predetermined layered data structures if $|\mathbf{R}_{i,j,k}|$ is less than said predetermined third limit value 12, otherwise the lookup is completed by a final SVN lookup using direct indexing in said fourth level.

11. The method according to claims 9 or 10, **characterized** in that in said second level the universe $\mathbf{U}$ is partitioned into 2^{16} sub-universes $\mathbf{U}_0, \mathbf{U}_1, \ldots, \mathbf{U}_{2^{16}-1}$, each of size 2^{16}, and in that the set of ranges $\mathbf{R}$ is processed into $\mathbf{R}_0, \mathbf{R}_1, \ldots, \mathbf{R}_{2^{16}-1}$ and represented by the sub-data structures $T_0, T_1, \ldots, T_{2^{16}-1}$.

12. The method according to claim 11, **characterized** in that in said third level each sub-universe $\mathbf{U}_i$ is partitioned into 2^8 sub-universes $\mathbf{U}_{i,0}, \mathbf{U}_{i,1}, \ldots, \mathbf{U}_{i,2^8-1}$, each of size 2^8, and in that the set of ranges $\mathbf{R}_i$ is processed into $\mathbf{R}_{i,0}, \mathbf{R}_{i,1}, \ldots, \mathbf{R}_{i,2^8-1}$ and represented by the sub-data structures $T_{i,0}, T_{i,1}, \ldots, T_{i,2^8-1}$.

13. The method according to claim 12, **characterized** in that in said fourth level each sub-universe $\mathbf{U}_{i,j}$ is partitioned into 2^4 sub-universes $\mathbf{U}_{i,j,0}, \mathbf{U}_{i,j,1}, .\mathbf{U}_{i,j,2^4-1}$, each of size 2^4, and in that the set of ranges $\mathbf{R}_{i,j}$ is processed into

$$\mathbf{R}_{i,j,0}, \mathbf{R}_{i,j,1}, \ldots, \mathbf{R}_{i,j,2^4-1}$$

and represented by the sub-data structures $T_{i,j,0}, T_{i,j,1}, \ldots, T_{i,j,2^4-1}$.

14. The method according to any of the claims 9 – 13, **characterized** in that said first set of data structures comprises Dynamic Flat Tree and Dynamic Layered

Tree structures.

15. The method according to any of the claims 9-- 14, **characterized** in that said second set of data structures comprises Dynamic Layered Tree structures.

16. The method according to any of the claims 9 -- 15, **characterized** in that said third set of data structures to be used in said fourth level of the lookup procedure comprises Dynamic Layered Tree and Static Flat Tree structures.

17. The method according to any of claims 1 -- 2 or 4 -- 6, **characterized** in that the maximum number of routes is fixed to 2187 and the maximum allowable number of memory accesses to 4, whereupon a 32 bits variation of the data structure Dynamic Layered Tree is used as construction block for the building of the forwarding table such that the memory needs equal 26240 bytes.

18. A system for IP routing lookup to determine where to forward an IP-datagram with a given destination address by retrieving from a routing table a next-hop index indicating where to forward said datagram, said next-hop index being associated with the longest matching prefix of said destination address, said address being a number in an address universe **U**, said system comprising a first converting means for converting a set of address prefixes **P** into a set of ranges **R** constituting a partition of said universe **U** and a second converting means for converting the mapping from **P** onto a set of next-hop indices D to an equivalent mapping from **R** onto D, **characterized** by data structuring means for forming predetermined layered data structures T representing the routing table, and building and memory means for building and storing a forwarding table representation from **R** and D by using a predetermined layered data structure where the construction of the layer is selected depending on the range density $|\mathbf{R}'|$ for the sub-universe $\mathbf{U}'$ represented by that layer to get a space efficient representation of the set of ranges **R**, and means for performing the lookup by a range matching operation in said forwarding table.

19. The system according to claim 18, **characterized** in that said forwarding table building and storing means are arranged to repeatedly reduces the size of the address sub-universes in successive levels a Static Vector Node (SVN) representation is used in a first level for the forwarding table representation, one of a

first set of said predetermined layered data structures, selectable depending on the actual value of the range density $|\mathbf{R}_i|$ to give the most efficient table representation, is chosen for building the forwarding table in a second level, if $|\mathbf{R}_i|$ is less than a predetermined first limit value, otherwise a SVN representation is used, in a third level one of a second set of said predetermined data structures, selectable depending on the actual value of the range density $|\mathbf{R}_{i,j}|$ to give the most efficient table representation, is chosen for building the forwarding table in this third level, if $|\mathbf{R}_{i,j}|$ is less than a predetermined second limit value, otherwise a SVN representation is used etc. until the building and storing is completed.

20. The system according to any of the claims 18 or 19, **characterized** in that said lookup performing means are arranged to perform an initial SVN lookup by direct indexing in said first level to retrieve a pointer that either represents a next-hop index, which means that the lookup is completed, or refers to a sub-data structure where the lookup is continued on said second level and, depending on the value of $|\mathbf{R}_i|$, encoded in the pointer value and obtained by decoding the pointer value, a lookup is completed in a forwarding table represented by one of said first set of said predetermined layered data structures if $|\mathbf{R}_i|$ is less than said predetermined first limit value, otherwise a SVN lookup by direct indexing is performed in said second level to retrieve a pointer that either represents a next-hop index, which means that the lookup is completed, or refers to a sub-data structure where the lookup is continued on said third level and, depending on the value of $|\mathbf{R}_{i,j}|$, encoded in the pointer value and obtained by decoding the pointer value, a lookup is completed in a forwarding table represented by one of said second set of said predetermined layered data structures if $|\mathbf{R}_{i,j}|$ is less than said predetermined second limit value, otherwise a SVN lookup by direct indexing is performed etc. until the lookup is completed.

21. The system according to any of the claims 18 – 20, **characterized** in that the maximum number of routes is fixed to 2^{18} routes, the maximum number of memory accesses for looking up the next-hop index is 4, and whereupon said data structures are selected as construction blocks for the building of the forwarding table in such a way that memory needs are less than 2.7 Mbytes.

22. The system according to any of the claims 18 – 21, **characterized** in that said forwarding table building and storing means are arranged to repeatedly reduces the size of the address sub-universes in successive levels a Static Vector Node (SVN) representation of size 2^{16} pointers is used in a first level for the forwarding table representation, one of a first set of said predetermined layered data structures, selectable depending on the actual value of the range density $|\mathbf{R}_i|$ to give the most efficient table representation, is chosen for building the forwarding table in a second level, if $|\mathbf{R}_i|$ is less than a predetermined first limit value 2313, otherwise a SVN representation of size 2^8 pointers is used, in a third level one of a second set of said predetermined data structures, selectable depending on the actual value of the range density $|\mathbf{R}_{i,j}|$ to give the most efficient table representation, is chosen for building the forwarding table in this third level, if $|\mathbf{R}_{i,j}|$ is less than a predetermined second limit value 137, otherwise a SVN representation of size 2^4 pointers is used, and in a fourth level one of a third set of said predetermined data structures, selectable depending on the actual value of the range density $|\mathbf{R}_{i,j,k}|$ to give the most efficient table representation, is chosen for building the forwarding table in this forth level, if $|\mathbf{R}_{i,j,k}|$ is less than a predetermined third limit value 12, otherwise a SVN representation of size 2^4 pointers is used.

23. The system according to any of the claims 18 – 22, **characterized** in that said lookup performing means are arranged to perform an initial SVN lookup by direct indexing in said first level to retrieve a pointer that either represents a next-hop index, which means that the lookup is completed, or refers to a sub-data structure where the lookup is continued on said second level and, depending on the value of $|\mathbf{R}_i|$, encoded in the pointer value and obtained by decoding the pointer value, a lookup is completed in a forwarding table represented by one of said first set of said predetermined layered data structures if $|\mathbf{R}_i|$ is less than said predetermined first limit value 2313, otherwise a SVN lookup by direct indexing is performed in said second level to retrieve a pointer that either represents a next-hop index, which means that the lookup is completed, or refers to a sub-data structure where the lookup is continued on said third level and, depending on the value of $|\mathbf{R}_{i,j}|$, encoded in the pointer value and obtained by decoding the pointer value, a lookup is completed in a forwarding table represented by

one of said second set of said predetermined layered data structures if $|\mathbf{R}_{i,j}|$ is less than said predetermined second limit value 137, otherwise a SVN lookup by direct indexing is performed in said third level to retrieve a pointer that either represents a next-hop index, which means that the lookup is completed, or refers to a sub-data structure where the lookup is continued on said fourth level and, depending on the value of $|\mathbf{R}_{i,j,k}|$, encoded in the pointer value and obtained by decoding the pointer value, a lookup is completed in a forwarding table represented by one of said third set of said predetermined layered data structures if $|\mathbf{R}_{i,j,k}|$ is less than said predetermined third limit value 12, otherwise the lookup is completed by a final SVN lookup using direct indexing in said fourth level.

24. The system according to any of the claims 21 – 23, **characterized** in that said data structuring means are adapted to partition, in a first level, the universe $\mathbf{U}$ into 2^{16} sub-universes

$$\mathbf{U}_0, \mathbf{U}_1, \ldots, \mathbf{U}_{2^{16}-1},$$

each of size 2^{16}, and to process the set of ranges $\mathbf{R}$ into

$$\mathbf{R}_0, \mathbf{R}_1, \ldots, \mathbf{R}_{2^{16}-1},$$

represented by the sub-data structures $T_0, T_1, \ldots, T_{2^{16}-1}$.

25. The system according to claim 24, **characterized** in that said data structuring means are adapted to partition, in a second level, each sub-universe $\mathbf{U}_i$ is into 2^8 sub-universes

$$\mathbf{U}_{i,0}, \mathbf{U}_{i,1}, \ldots, \mathbf{U}_{i,2^8-1},$$

each of size 2^8, and to process the set of ranges $\mathbf{R}_i$ into

$$\mathbf{R}_{i,0}, \mathbf{R}_{i,1}, \ldots, \mathbf{R}_{i,2^8-1},$$

represented by the sub-data structures $T_{i,0}, T_{i,1}, \ldots, T_{i,2^8-1}$.

26. The system according to claim 25, **characterized** in that said data structuring means are adapted to partition, in a third level, each sub-universe $\mathbf{U}_{i,j}$ into 2^4 sub-universes

$$\mathbf{U}_{i,j,0}, \mathbf{U}_{i,j,1}, \ldots, \mathbf{U}_{i,j,2^4-1},$$

each of size 2^4, and to process the set of ranges $\mathbf{R}_{i,j}$ into

$$\mathbf{R}_{i,j,0}, \mathbf{R}_{i,j,1}, \ldots, \mathbf{R}_{i,j,2^4-1},$$

represented by the sub-data structures $T_{i,j,0}, T_{i,j,1}, \ldots, T_{i,j,2^4-1}$.

27. The system according to any of the claims 18 – 26, **characterized** in that said data structuring means are adapted to form data structures comprising Dynamic Flat Tree, Dynamic Layered Tree and Static Flat Tree data structures.

28. The system according to any of the claims 18 – 20, **characterized** in that said data structuring means is adapted to form a 32 bits variation of a Dynamic Layered Tree data structure as construction block for building of a forwarding table such that memory needs equal 26240 bytes for a maximum number of routes of 2187 and a maximum number of memory accesses of 4.

29. A computer program product, characterized by computer program code means to make a computer execute the method according to any of the claims 1 – 17 when the program is run on a computer.

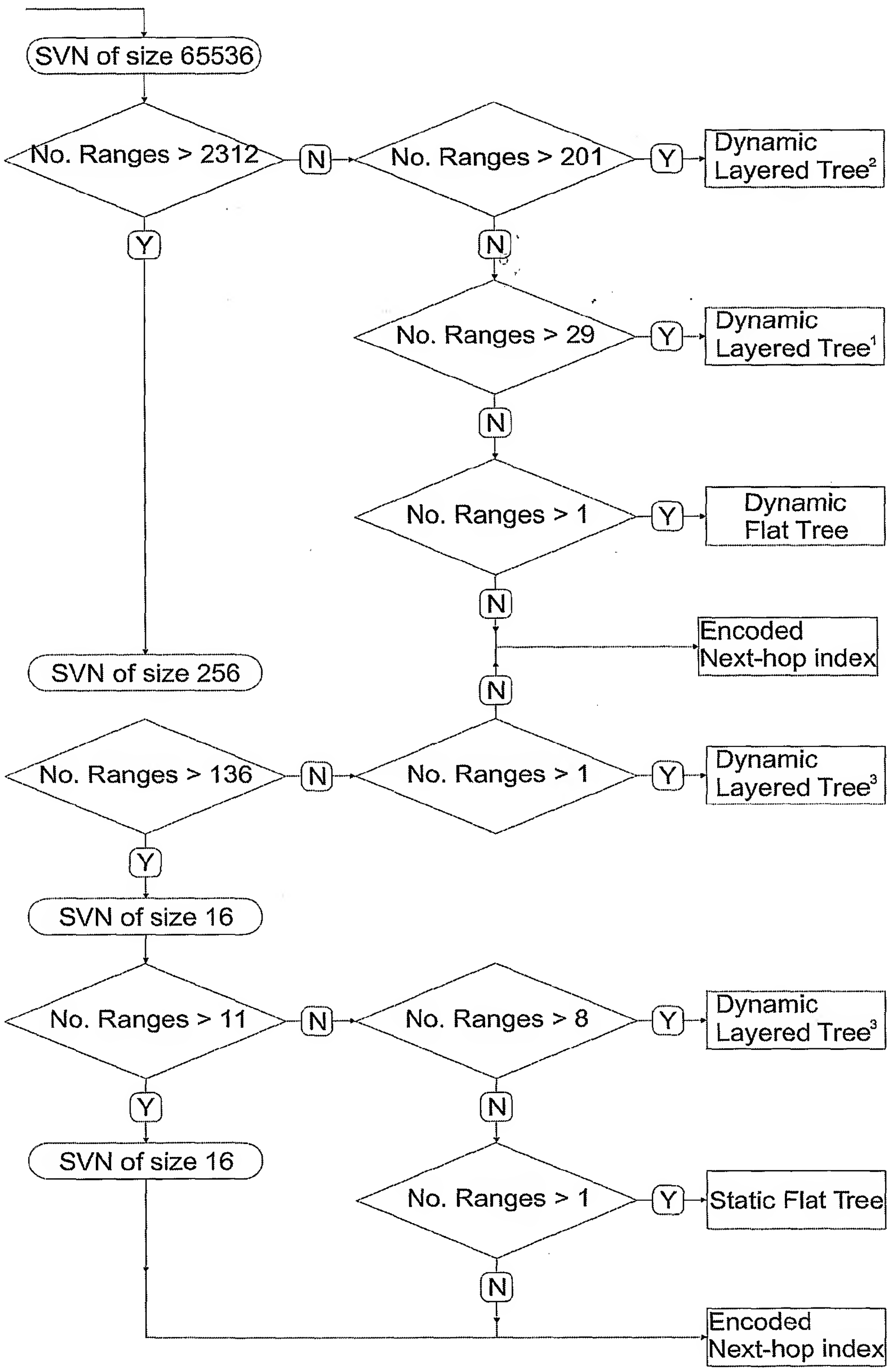

SVN of size 65536
No. Ranges > 2312
N
No. Ranges > 201
Y
Dynamic Layered Tree²
Y
N
No. Ranges > 29
Y
Dynamic Layered Tree¹
N
No. Ranges > 1
Y
Dynamic Flat Tree
N
Encoded Next-hop index
SVN of size 256
N
No. Ranges > 136
N
No. Ranges > 1
Y
Dynamic Layered Tree³
Y
SVN of size 16
No. Ranges > 11
N
No. Ranges > 8
Y
Dynamic Layered Tree³
Y
N
SVN of size 16
No. Ranges > 1
Y
Static Flat Tree
N
Encoded Next-hop index

Figure 1

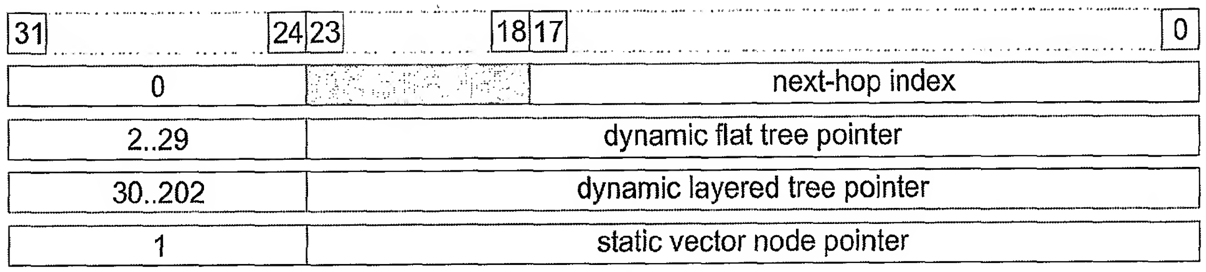

31
24
23
18
17
0
0
next-hop index
2..29
dynamic flat tree pointer
30..202
dynamic layered tree pointer
1
static vector node pointer

Figure 2

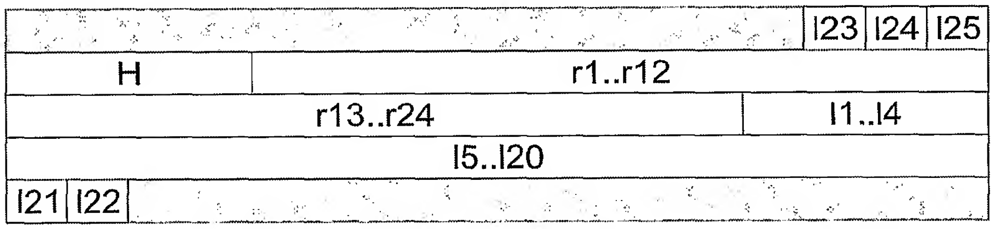


Figure 3

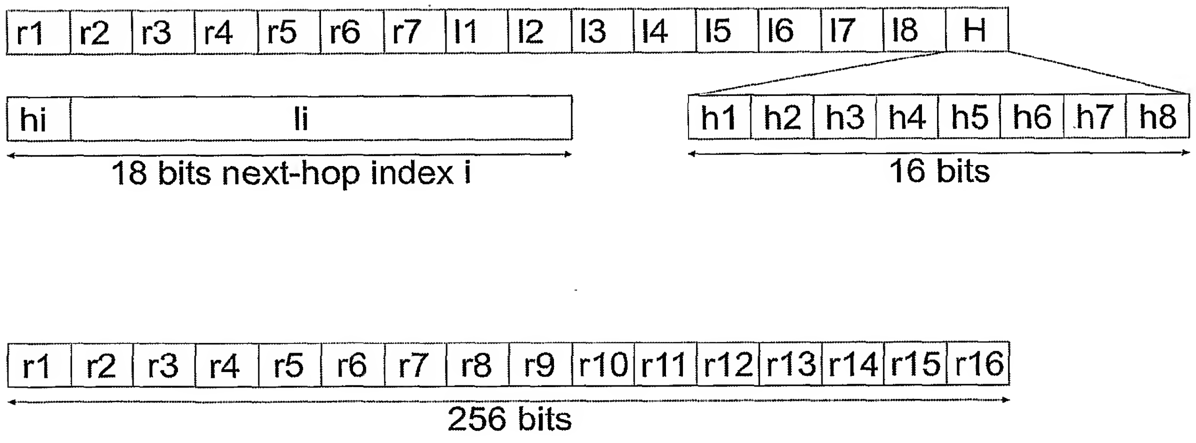


Figure 4

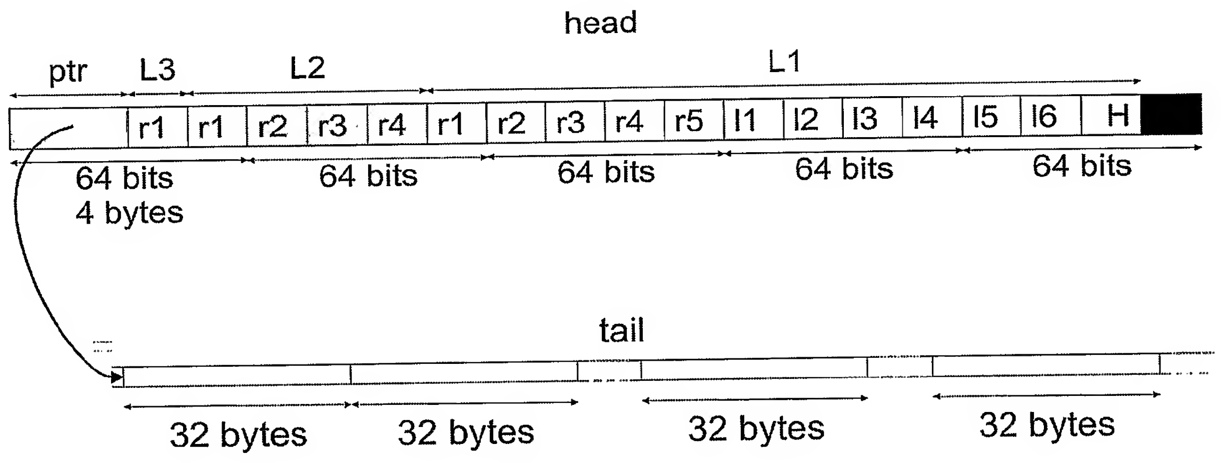


Figure 5

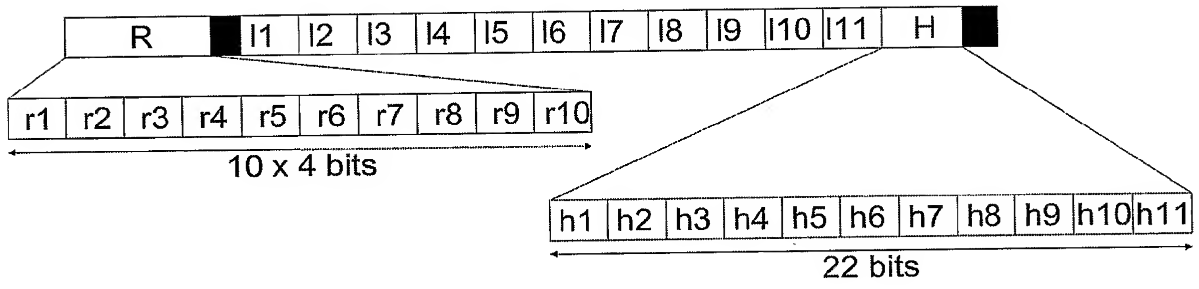


Figure 6

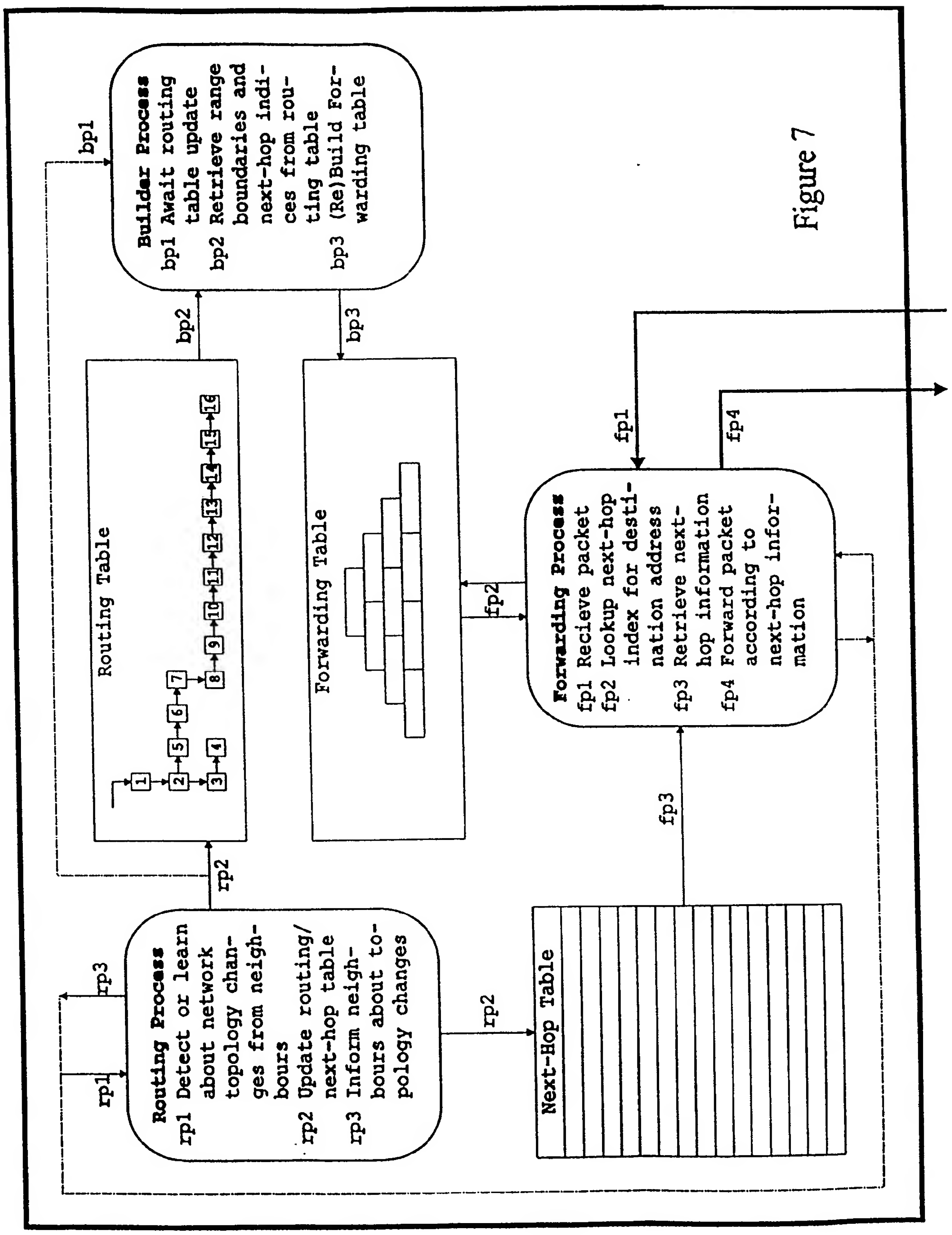

Builder Process
bp1 Await routing table update
bp2 Retrieve range boundaries and next-hop indices from routing table
bp3 (Re)Build Forwarding table
bp1
bp2
bp3
Routing Table
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Forwarding Table
Forwarding Process
fp1 Recieve packet
fp2 Lookup next-hop index for destination address
fp3 Retrieve next-hop information
fp4 Forward packet according to next-hop information
fp1
fp2
fp3
fp4
Routing Process
rp1 Detect or learn about network topology changes from neighbours
rp2 Update routing/ next-hop table
rp3 Inform neighbours about topology changes
rp1
rp2
rp3
Next-Hop Table

Figure 7

INFINITY	INFINITY	INFINITY	INFINITY	INFINITY	INFINITY		INFINITY		INFINITY	
INFINITY	INFINITY	INFINITY	INFINITY	INFINITY	INFINITY		INFINITY		INFINITY	
0576330784	2797273088	3363101952	4126764016	INFINITY	INFINITY		INFINITY		INFINITY	
0000000666	0000000667	0050331648	0054525952	0268435456	00009	00016	00009	00007	00009	00001
0576330816	2108276736	2108293120	2147483648	2684354560	00015	00001	00012	00001	00010	00004
2797289472	3083878400	3083894784	3295051776	3295084544	00003	00004	00005	00004	00014	00004
3363102080	4049862656	4049928192	4074831872	4074864640	00006	00004	00002	00004	00013	00004
4126764032	4256393216	4256395264	INFINITY	INFINITY	00008	00004	00011	00004	EMPTY	EMPTY

Figure 8

INTERNATIONAL SEARCH REPORT

International application No.
PCT/SE 03/00064

A. CLASSIFICATION OF SUBJECT MATTER

IPC7: H04L 12/56, H04Q 11/04

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

IPC7: H04L, H04Q, G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

SE,DK,FI,NO classes as above

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

EPO-INTERNAL, WPI DATA, PAJ

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	WO 0122667 A1 (EFFNET GROUP AB), 29 March 2001 (29.03.01), page 1, line 6 - page 3, line 14, claims 1-27, abstract --	1-29
A	WO 9914906 A1 (EFFICIENT NETWORKING AB), 25 March 1995 (25.03.95), page 1, line 6 - page 5, line 35, claims 1-2, abstract --	1-29
A	WO 0137495 A1 (BROADCOM CORP), 25 May 2001 (25.05.01), page 1, line 12 - page 4, line 3, claims 1-15, abstract --	1-29

[X] Further documents are listed in the continuation of Box C. [X] See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance: the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance: the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search 2 April 2003	Date of mailing of the international search report 04-04-2003
Name and mailing address of the ISA/ Swedish Patent Office Box 5055, S-102 42 STOCKHOLM Facsimile No. +46 8 666 02 86	Authorized officer Roger Bou Faisal /LR Telephone No. +46 8 782 25 00

INTERNATIONAL SEARCH REPORT

International application No.
PCT/SE 03/00064

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	ERGUN; F: et al: A Dynamic Lookup Scheme for Bursty Access Patterns. 2001-01-12, IEEE. See whole document --	1-29
A	WO 9913619 A2 (SICS SWEDISH INSTITUTE OF COMPUTER SCIENCE), 18 March 1999 (18.03.99), claims 1-14, abstract -- --------	1-29

INTERNATIONAL SEARCH REPORT

Information on patent family members

28/02/03

International application No.
PCT/SE 03/00064

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 0122667 A1	29/03/01	AU 7819300 A	24/04/01
		SE 9903460 A	23/03/01
WO 9914906 A1	25/03/95	AU 7463198 A	05/04/99
		BG 104202 A	31/10/00
		CA 2303118 A	25/03/99
		CN 1270728 T	18/10/00
		EE 200000228 A	15/06/01
		EP 1016245 A	05/07/00
		IL 134835 D	00/00/00
		JP 2001517024 T	02/10/01
		NO 20001309 A	02/05/00
		PL 339268 A	04/12/00
		SE 9703332 D	00/00/00
		SK 3692000 A	12/09/00
		US 6266706 B	24/07/01
WO 0137495 A1	25/05/01	AU 1754801 A	30/05/01
		EP 1232612 A	21/08/02
WO 9913619 A2	18/03/99	AU 9100898 A	29/03/99
		CA 2302744 A	18/03/99
		EP 1025678 A	09/08/00
		SE 511972 C	10/01/00
		SE 9703292 A	10/03/99